CodeWarrior™ Development Studio Motorola® DSP56800E Embedded Systems Targeting Manual

CWDSP56800E1/D
REV: 1
10/2002

Revised: 20021021

Metrowerks, the Metrowerks logo, and CodeWarrior are registered trademarks of Metrowerks Corp. in the US and/or other countries. All other tradenames and trademarks are the property of their respective owners.

© Copyright. 2002. Metrowerks Corp. ALL RIGHTS RESERVED.

Metrowerks reserves the right to make changes to any product described or referred to in this document without further notice. Metrowerks makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Metrowerks assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft or automobile navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.

Documentation stored on electronic media may be printed for non-commercial personal use only, further to the license agreement related to the product associated with the documentation. Subject to the foregoing non-commercial personal use, no portion of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks.

USE OF ALL SOFTWARE, DOCUMENTATION AND RELATED MATERIALS ARE SUBJECT TO THE METROWERKS END USER LICENSE AGREEMENT FOR SUCH PRODUCT.

How to Contact Metrowerks:

Corporate Headquarters	Metrowerks Corporation 9801 Metric Blvd. Austin, TX 78758 U.S.A.
World Wide Web	`http://www.metrowerks.com`
Ordering & Technical Support	Voice: (800) 377-5416 Fax: (512) 997-4901

Table of Contents

1 Introduction **15**

CodeWarrior IDE and Its Documentation 15
New In This Release 17
References 17

2 Getting Started **19**

System Requirements 19
DSP56800E Hardware Requirements 19
Installing the CodeWarrior IDE for DSP56800E 20
Installing the CodeWarrior IDE 20
What Gets Installed. 23
Creating a Project 24
Editing the Contents of a Project 27

3 Development Studio Overview **29**

CodeWarrior IDE 29
CodeWarrior Compiler for DSP56800E 30
CodeWarrior Assembler for DSP56800E 30
CodeWarrior Linker for DSP56800E. 30
CodeWarrior Debugger for DSP56800E 30
Metrowerks Standard Library 31
The Development Process 31
Project Files versus Makefiles 31
Editing Code 32
Compiling. 32
Linking 34
Debugging 34
Viewing Preprocessor Output 34

4 Target Settings **35**

Target Settings Overview 35
Target Setting Panels 36
Changing Target Settings 37
Exporting and Importing Panel Options to XML Files 38
Exporting Panel Options to XML File 38

Importing Panel Options from XML File 38
Saving New Target Settings in Stationery 38
Restoring Target Settings 39
CodeWarrior IDE Target Settings Panels 40
DSP56800E-Specific Target Settings Panels 41
Target Settings 41
M56800E Target 43
C/C++ Language 45
C/C++ Warnings. 51
M56800E Assembler 55
ELF Disassembler 58
M56800E Processor 61
The M56800E Processor panel options are: 62
M56800E Linker 65
Remote Debugging 70
Remote Debug Options 72
M56800E Target 74

5 C for DSP56800E 79

General Notes on C 79
Number Formats for DSP56800E 80
56800E Integer Formats 80
56800E Floating-Point Formats 81
Calling Conventions and Stack Frames for DSP56800E. 82
Passing Parameters to Functions 82
Returning Values From Functions 83
Volatile and Non-Volatile Registers 83
Non-volatile Registers 83
Volatile Registers 83
Stack Frame 86
Stack Alignment 87
Data Alignment Requirements for DSP56800E 87
Pointer Types (Word versus Byte Pointers). 88
Reordering of Data for Optimal Usage 89
Code and Data Storage for DSP56800E 89
Additional Information about Character Data Addressing . . 90
Large Data Model Support 91

Examples: Extended Data Addressing 92
Examples: Accessing Data Objects 93
Example 1 93
Example 2 94
External Library Compatibility 94
Optimizing Code for DSP56800E 95
Pragma Directives for DSP56800E 97
asmoutput 97
check_c_src_pipeline 98
check_inline_asm_pipeline 98
interrupt 98
Arguments 99
Avoiding Possible Hitches with enabled Pragma Interrupt 103
optimization_level 104
optimize_for_size 105
define_section 106
section 108
unused 109
Linker Issues for DSP56800E 109
Deadstripping Unused Code and Data 109
Link Order 110

6 Inline Assembly Language and Intrinsics 111

Working With DSP56800E Inline Assembly Language 111
Inline Assembly Language Syntax for DSP56800E 112
Function-level Inline Assembly Language 112
Statement-level Inline Assembly Language 112
Adding Assembly Language to C Source Code 113
Assembly Language Quick Guide 114
Creating Labels for M56800E Assembly 114
Using Comments in M56800E Assembly 115
Inline Assembly Optimization 115
Calling Assembly Language Functions from C Code 115
Calling Inline Assembly Language Functions 115
Calling Pure Assembly Language Functions 116
Calling Functions from Assembly Language 119
Instrinsic Functions for DSP56800E 120

An Overview of Intrinsic Functions 120
Implementation 121
Fractional Arithmetic 122
Macros Used with Intrinsics 122
Intrinsic Functions for Math Support 123
Absolute/Negate. 124
abs_s 124
negate 124
L_abs 125
L_negate 125
Addition/Subtraction. 126
add 126
sub 127
L_add 127
L_sub 128
Control 129
stop 129
wait 129
turn_off_conv_rndg 130
turn_off_sat 130
turn_on_conv_rndg 130
turn_on_sat. 131
Deposit/ Extract 132
extract_h 132
extract_l 132
L_deposit_h. 133
L_deposit_l. 133
Division. 134
div_s 134
div_s4q 135
div_ls. 135
div_ls4q 136
Multiplication/ MAC 137
mac_r 137
msu_r 138
mult 138
mult_r. 139

L_mac 139
L_msu 140
L_mult 140
L_mult_ls 141
Normalization 142
ffs_s 142
norm_s 143
ffs_l 143
norm_l 144
Rounding 145
round 145
Shifting 146
shl 147
shlftNs 148
shlfts 149
shr 150
shr_r 151
shrtNs 152
L_shl 153
L_shlftNs 153
L_shlfts 154
L_shr 155
L_shr_r 156
L_shrtNs 156
Modulo Addressing Intrinsic Functions 157
An Overview of Modulo Addressing 157
Modulo Addressing Intrinsic Functions: Definitions and Examples 158
__mod_init 159
__mod_initint16 160
__mod_start 160
__mod_access 160
__mod_update 161
__mod_stop 161
__mod_getint16 161
__mod_setint16 162
__mod_error 162

Modulo Buffer Examples 163
Caveats on Using the Modulo Buffer API 165
Error Code Documentation 166

7 Debugging for DSP56800E 169

Target Settings for Debugging 169
Command Converter Server 170
Essential Target Settings for Command Converter Server . . 171
Changing the Command Converter Server Protocol to Parallel Port 172
Changing the Command Converter Server Protocol to PCI. . 174
Setting Up a Remote Connection 174
To Add a New Remote Connection 175
To Change an Existing Remote Connection 177
To Remove an Existing Remote Connection 177
Debugging a Remote Target Board 177
Load/Save Memory 177
History Combo Box 178
Radio Buttons 179
Memory Type Combo Box 179
Address Text Field 179
Size Text Field 179
Dialog Box Controls 179
Fill Memory 180
History Combo Box 180
Memory Type Combo Box 181
Address Text Field 181
Size Text Field 181
Fill Expression Text Field 181
Dialog Box Controls 182
Save/Restore Registers 182
History Combo Box 183
Radio Buttons 183
Register Group List 184
Dialog Box Controls 184
EOnCE Debugger Features. 184
Set Hardware Breakpoint Panel 185

Special Counters 186
Trace Buffer 187
Set Trigger Panel 190
Using the DSP56800E Simulator 193
Cycle/Instruction Count 193
Machine Cycle Count. 193
Machine Instruction Count 194
Launching and Operating the Debugger 194
Setting Breakpoints 197
Setting Watchpoints 198
Viewing and Editing Register Values 199
Register Details Window 201
Viewing X: Memory 202
Viewing P: Memory 203
Loading a .elf File without a Project 207
Command-Line Debugging 208
Tcl Support 208
Automatically resolving clashing commands 208
Tcl support for executing script files 209
Tcl support for using a start-up script 209
Command-Line Debugging Tasks 210
Open a Command-Line Debugging Window 210
Enter a Single Command-Line Debugging Command. . . 210
Enter Multiple Command-Line Debugging Commands . . 210
View Debugging Command Hints 211
Repeat a Command-Line Debugging Command 211
Review Previously Entered Commands 211
Clear a Command from the Command Line 212
Stop an Executing Script 212
Switch between Insert and Overwrite Mode 212
Scroll Text in the Command-Line Debugging Window . . 212
Copy Text from the Command-Line Debugging Window . 213
Paste Text into the Command-Line Debugging Window. . 213
Command-Line Debugging Commands 213
alias 213
break 214
bringtofront 215

cd 216
change 217
cls 219
close 219
config 219
copy 221
debug 222
dir or ls 222
disassemble 223
display 224
evaluate 226
exit 227
go 227
help 228
history 228
hsst_attach_listener 229
hsst_block_mode 229
hsst_close 230
hsst_detach_listener 230
hsst_log 230
hsst_noblock_mode 231
hsst_open 231
hsst_read 231
hsst_write 232
input 232
kill 233
load 234
log 235
next 236
output 236
pwd 237
radix 237
restart 239
run 239
save 240
step 241
stop 242

switchtarget 243
system 243
view 244
wait 244
watchpoint 245
System-Level Connect 245
Debugging in the Flash Memory 246
Flash Memory Commands 246
set_hfmclkd <value> 246
set_hfm_base <address> 247
set_hfm_config_base <address> 247
add_hfm_unit <startAddr> <endAddr> <bank> <numSectors> <pageSize> <progMem> <boot> <interleaved> 247
set_hfm_erase_mode units | pages | all 248
set_hfm_verify_erase 1 | 0 248
set_hfm_verify_program 1 | 0 248
Unlock Flash Security Features 248
Notes for Debugging on Hardware 248

8 High-Speed Simultaneous Transfer 251

Host-Side Client Interface 251
hsst_open 252
hsst_close 252
hsst_read 253
hsst_write 254
hsst_size 255
hsst_block_mode 255
hsst_noblock_mode 256
hsst_attach_listener 256
hsst_detach_listener 257
hsst_set_log_dir 257
HSST Host Program Example 258
Target Library Interface 259
HSST_open 260
HSST_close 260
HSST_setvbuf 260

HSST_write 262
HSST_read 263
HSST_flush 263
HSST_size 264
HSST_raw_read 264
HSST_raw_write 265
HSST_set_log_dir 266
HSST Target Program Example 266

9 ELF Linker and Command Language 269

Structure of Linker Command Files 270
Memory Segment 270
Closure Blocks 271
Sections Segment 272
Linker Command File Syntax 273
Alignment 273
Arithmetic Operations 274
Comments 274
Deadstrip Prevention 275
Variables, Expressions, and Integral Types 275
Variables and Symbols 275
Expressions and Assignments 276
Integral Types 276
File Selection 278
Function Selection 279
ROM to RAM Copying 279
Stack and Heap 281
Writing Data Directly to Memory 282
Linker Command File Keyword Listing 283
. (location counter) 283
ADDR 284
ALIGN 284
ALIGNALL 285
FORCE_ACTIVE 285
GROUP 286
INCLUDE 286
KEEP_SECTION 286

MEMORY 287
OBJECT 289
REF_INCLUDE 289
SECTIONS 289
SIZEOF 291
SIZEOFW 291
WRITEB 291
WRITEH 292
WRITEW 292
DSP56800E Command-Line Tools 293
Usage 293
Response File 294
Sample Build Script 295
Arguments 296
General Command-Line Options 296
Compiler 297
Linker 304
Assembler 307

10 Libraries and Runtime Code 309

MSL for DSP56800E 309
Using MSL for DSP56800E 309
Console and File I/O 310
Allocating Stacks and Heaps for the DSP56800E 312
Definitions 313
Runtime Initialization 315
EOnCE Library 318
_eonce_Initialize 320
_eonce_SetTrigger 321
_eonce_SetCounterTrigger 322
_eonce_ClearTrigger 324
_eonce_GetCounters 325
_eonce_GetCounterStatus 326
_eonce_SetupTraceBuffer 327
_eonce_GetTraceBuffer 328
_eonce_ClearTraceBuffer 329
_eonce_StartTraceBuffer 330

_eonce_HaltTraceBuffer 331
_eonce_EnableDEBUGEV 332
_eonce_EnableLimitTrigger 333
Definitions 334
Return Codes 334
Normal Trigger Modes 335
Counter Trigger Modes 336
Data Selection Modes. 338
Counter Function Modes 339
Normal Unit Action Options 339
Counter Unit Action Options 340
Accumulating Trigger Options. 340
Miscellaneous Trigger Options. 342
Trace Buffer Capture Options 342
Trace Buffer Full Options 343
Miscellaneous Trace Buffer Option 344

Index **345**

1

Introduction

This manual explains how to use the CodeWarrior™ Integrated Development Environment (IDE) to develop code for the DSP56800E family of processors.

This chapter contains the following sections:

- CodeWarrior IDE and Its Documentation
- References

CodeWarrior IDE and Its Documentation

The CodeWarrior IDE has a simple graphical user interface and versatile tools for developing software. Using the CodeWarrior IDE, you can develop a program, plug-in, library, or other code.

The CodeWarrior IDE allows you to assemble source-code files, resource files, library files, and configuration settings in a project without writing a complicated build script or makefile. You can also add or delete source-code files from a project using the mouse and keyboard instead of tediously editing a build script.

For any project you can create and manage several configurations for use on different computer platforms. The platform on which you run the CodeWarrior IDE is called the host. From that host, you use the CodeWarrior IDE to develop code to target various platforms.

The term target has two meanings in the CodeWarrior IDE:

- **Platform Target**

  The operating system, processor, or microcontroller for which you write code. If you write code for a particular processor running a specific desktop operating system, you are creating code for a platform target.

- **Build Target**

  The settings and files that determine the contents of your code, as well as the process in which this code compiles and links into the final output.

The CodeWarrior IDE lets you specify multiple build targets for a particular platform target. For example, you can compile a debugging version (build target) and an optimized version (build target) of your program for the Windows® operating system (platform target). The build targets can share files in the same project even though each build target uses its own settings. After debugging the program, generating a final version is as simple as selecting the project's build target and using a single `Make` command.

The CodeWarrior IDE has compilers, linkers, a debugger, a source-code browser, and editing tools. You can edit, navigate, examine, compile, link, and debug code all within the CodeWarrior IDE. Options for code generation, debugging, and navigation of your project are all configurable in the CodeWarrior IDE.

Most features of the CodeWarrior IDE apply to several hosts, languages, and build targets. However, each build target has its own unique features. This manual explains those features unique to the CodeWarrior IDE for the DSP56800E.

For a complete understanding of the CodeWarrior IDE, refer to the CodeWarrior IDE User's Guide for general information and this manual for information specific to the DSP56800E processor.

NOTE The CodeWarrior Release Notes contain information about new features, bug fixes, and incompatibilities that do not appear in the documentation because of release deadlines. Release Notes are on the CodeWarrior IDE CD.

New In This Release

Additions in this version are:

- Large data memory model support
- Inline 32-bit multiply routines
- Emit separate character section option in preference panel
- #pragma interrupt [comr] option
- #pragma optimize_for_size on | off | reset option

References

- The following manuals are included with this product:
  - *Code Warrior IDE User's Guide*
  - *Assembler Reference Manual*
  - *MSL C Reference* (Metrowerks Standard C libraries)
- To learn more about the DSP56800E processor, refer to the following manual:
  - *DSP56800E Family Manual.* Motorola, Inc., 2001

To download electronic copies of these manuals or order printed versions, visit this web address:

`http://www.motorola.com/`

2

Getting Started

This chapter explains how to install and run the CodeWarrior™ IDE on your Windows® operating system. This chapter also explains how to connect your hardware for each of the communications protocols supported by the CodeWarrior debugger.

This chapter contains the following sections:

- System Requirements
- Installing the CodeWarrior IDE for DSP56800E
- Creating a Project
- Editing the Contents of a Project

System Requirements

This section lists the system requirements to operate the CodeWarrior IDE for DSP56800E.

Table 2.1 lists requirements for installing and operating the CodeWarrior IDE.

Table 2.1 Requirements for the CodeWarrior IDE

Category	Requirement
Hardware	133 MHz Pentium® compatible processor 64 MB RAM CD-ROM drive for installation
Software	Microsoft® Windows® 98/2000/NT/XP
Other	200 MB hard drive space

DSP56800E Hardware Requirements

The hardware requirements necessary to debug the DSP56800E are:

- Power Supply
- 56800E EVM or Custom 56800E Development Board with a JTAG header connected

Installing the CodeWarrior IDE for DSP56800E

The CodeWarrior installer automatically installs all the necessary components for you to begin development. You must use the CodeWarrior installer to ensure that you have all the required files. If you have any questions regarding the installer, read the instructions provided in the CodeWarrior CD.

Installing the CodeWarrior IDE

To install the CodeWarrior IDE, perform these steps:

1. **Insert the CodeWarrior CD into your computer's CD-ROM drive.**

   If Auto Install is disabled on your computer, you must run the `setup.exe` program at the root directory on the CD.

2. **Follow the CodeWarrior software installation instructions on your screen.**

   After installing the CodeWarrior IDE, restart your computer. Restarting ensures that the newly installed drivers are available for use.

3. **Registered and 30-day evaluation users have separate registering procedures:**

   - Registered Users

     The Register CodeWarrior program (`MWRegister.exe`) is run as part of the installation procedure.

     However, if it is not run, you can run `MWRegister.exe` located in the licensing directory, and follow the directions on the screen.

     If you have any licensing questions, please contact: `license@metrowerks.com`.

- 30-day Evaluation Users

  Visit the following website and enter your validation code (located on the CodeWarrior Studio CD case):

  `http://www.metrowerks.com/key/eval/`

  You will receive a license key by email after you submit your information in the evaluation form at this website.

4. **Install the license key as follows:**

   a. **Locate the `license.dat` file in your CodeWarrior installation folder.**

      The file is located at the root of the CodeWarrior installation directory, so if you have installed the CodeWarrior software at the default installation path, you can find it here:

      `C: \Program Files\Metrowerks\ CodeWarrior\license.dat`

      It is important that this file remain at this location.

   b. **Use any standard text editor to open this file.**

      For example, NotePad.

   c. **Copy or type the key starting on a new line at the bottom of this file.**

      For example, if your new license key was:

```
FEATURE Win32_Plugins_DSP56800E metrowks 2.000 permanent uncounted\
         5FC07C47D6FB HOSTID=00c04ff5efb0
```

And the existing license.dat file contained:

```
FEATURE Win32_CWIDE_Limited metrowks 5.0 permanent uncounted \
         2C3C43468173 HOSTID=ANY
FEATURE Win32_CWIDE_Unlimited metrowks 5.0 permanent uncounted \
         D8C287BC5B1B HOSTID=ANY
```

After pasting or typing in the new key, the file contains:

```
FEATURE Win32_CWIDE_Limited metrowks 5.0 permanent uncounted \
         2C3C43468173 HOSTID=ANY
FEATURE Win32_CWIDE_Unlimited metrowks 5.0 permanent uncounted \
         D8C287BC5B1B HOSTID=ANY
FEATURE Win32_Plugins_DSP56800E metrowks 2.000 permanent  uncounted \
         5FC07C47D6FB HOSTID=00c04ff5efb0
```

d. Any future keys can likewise be appended to the bottom of this file.

If you encounter difficulty in installing this key, please contact Metrowerks Customer support at:

Ph: (800) 377-5416

Fax: (512) 873-4901

email: license@metrowerks.com

NOTE Do not move the license.dat file after installation.

What Gets Installed

Table 2.2 describes the folders that are installed as part of the standard full CodeWarrior IDE for the DSP56800E installation. Each of the folders is located in your CodeWarrior IDE installation directory.

Table 2.2 Contents Installed with the CodeWarrior IDE for DSP56800E

Directory Name	Contents
Bin	The CodeWarrior IDE application and associated plug-in tools.
CCS	Command converter server executable files and related support files.
CodeWarrior Examples	Target specific projects and code.
CodeWarrior Help	All the core help files for the CodeWarrior IDE, as well as target specific help. All help files are accessed through the Help menu or F1 help.
CodeWarrior Manuals	The CodeWarrior documentation tree.
CW Release Notes	Release notes for the CodeWarrior IDE and each tool.
Licensing	The registration program and additional licensing information.
M56800E_EABI_Tools	Drivers for the CCS and command line tools. Additional default files used by the CodeWarrior IDE for the DSP56800E stationery.
M56800E Support	Includes Metrowerks Standard Library (MSL), a subset of the standard MSL adapted specifically for DSP56800E.
Motorola Documentation	Documentation specific to the Motorola DSP568xxE series.
Stationery	Default settings that are used to create DSP56800E projects. Each target stationery item is set to a specific debugging protocol.

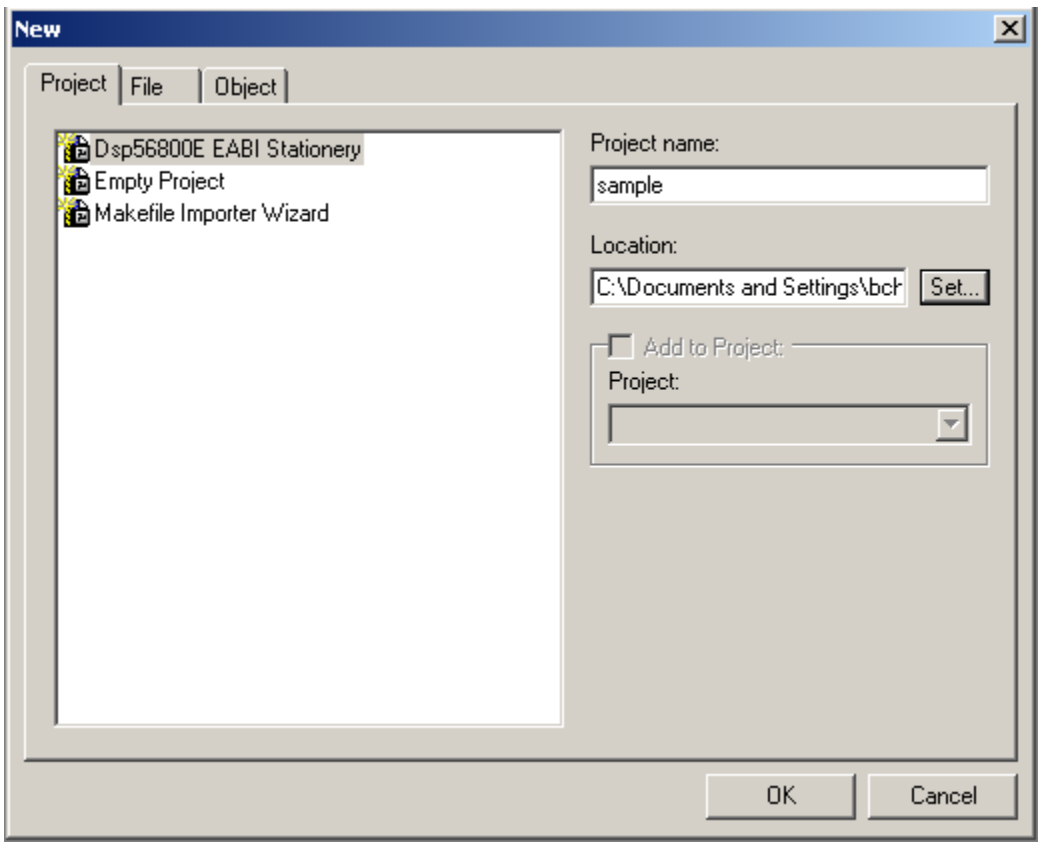
Creating a Project

To create a new project from project stationery:

1. **From the menu bar of the Metrowerks CodeWarrior window, select File > New.**

   The **New** window appears with a list of options in the **Projects** tab (Figure 2.1).

Figure 2.1 New Window



2. **Select DSP56800E EABI Stationery in the Project tab.**

NOTE To create a new project without using stationery, select **Empty Project** in the **New** window. This option lets you create a project from scratch. If you are a beginner, do not use an empty project because of the complexities involved in using the correct libraries and files and selecting the correct build target settings.

3. **Type a name in the Project name field.**

   The CodeWarrior IDE adds the `.mcp` extension automatically to your file when the project is saved. The `.mcp` extension allows any user to recognize the file as a Metrowerks CodeWarrior project file. For example, DSP56800E.mcp.

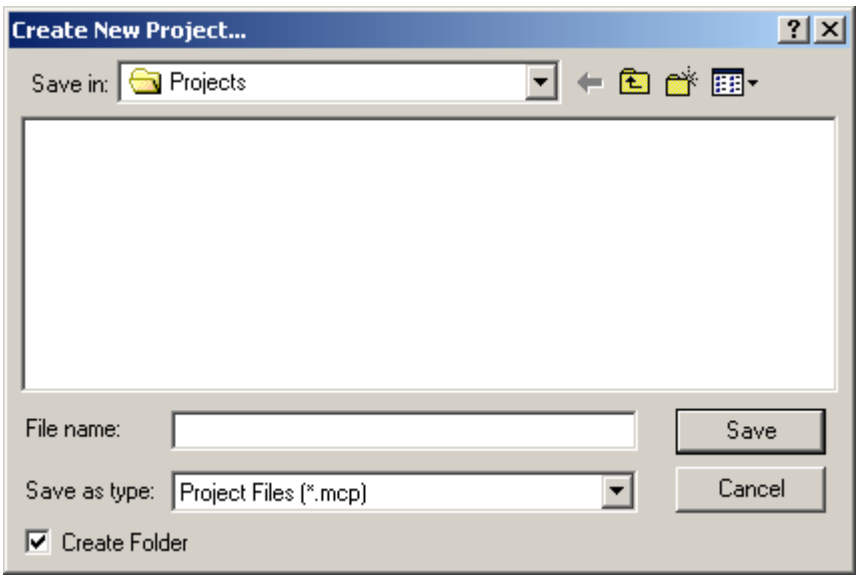
4. **Set the location for the project use.**

   If you want to change the default location, perform the following steps:

5. **Click the Set button.**

   The **Create New Project** dialog box (Figure 2.2) appears:

Figure 2.2 Create New Project Dialog Box



   a. **Specify the path where you want the project file to be saved.**

      Use the standard navigation controls in the **Create New Project** dialog box.

   b. **Click the Save button.**

      The CodeWarrior IDE closes the **Create New Project** dialog box.

   If you want to use the default location for your project, go to step 5.

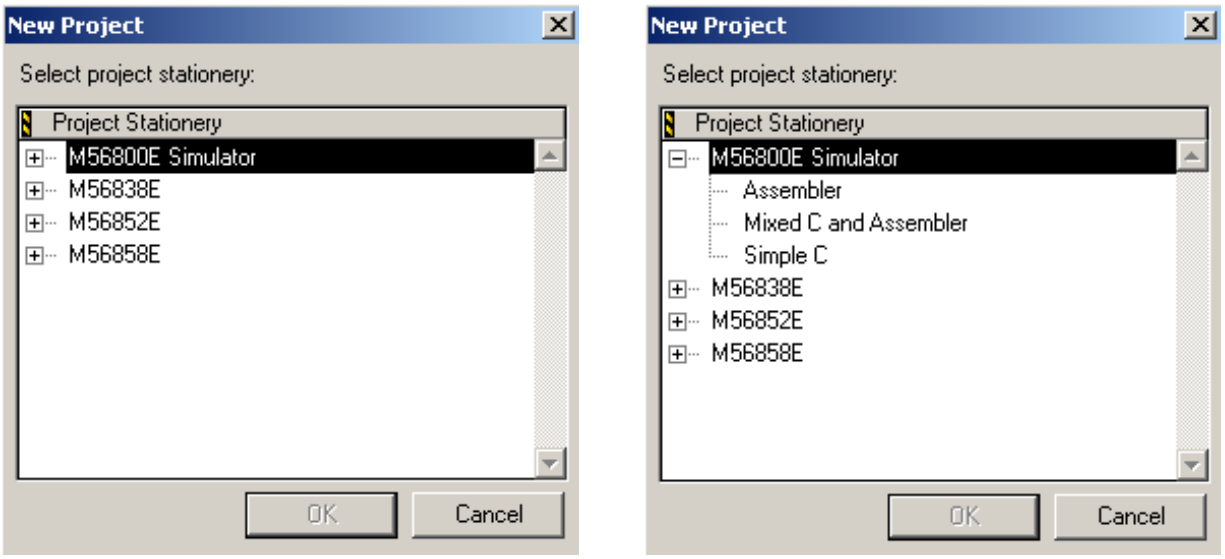
   In either case, the CodeWarrior IDE creates a folder with the same name as your project in the directory you select.

NOTE Enable the **Create Folder** checkbox in the **Create New Project** file dialog box to create a new folder for your project in the selected location.

6. **Click OK in the New window.**

   The **New Project** window appears (Figure 2.3) with a list of board-specific project stationeries.

Figure 2.3 New Project Window



(a) (b)

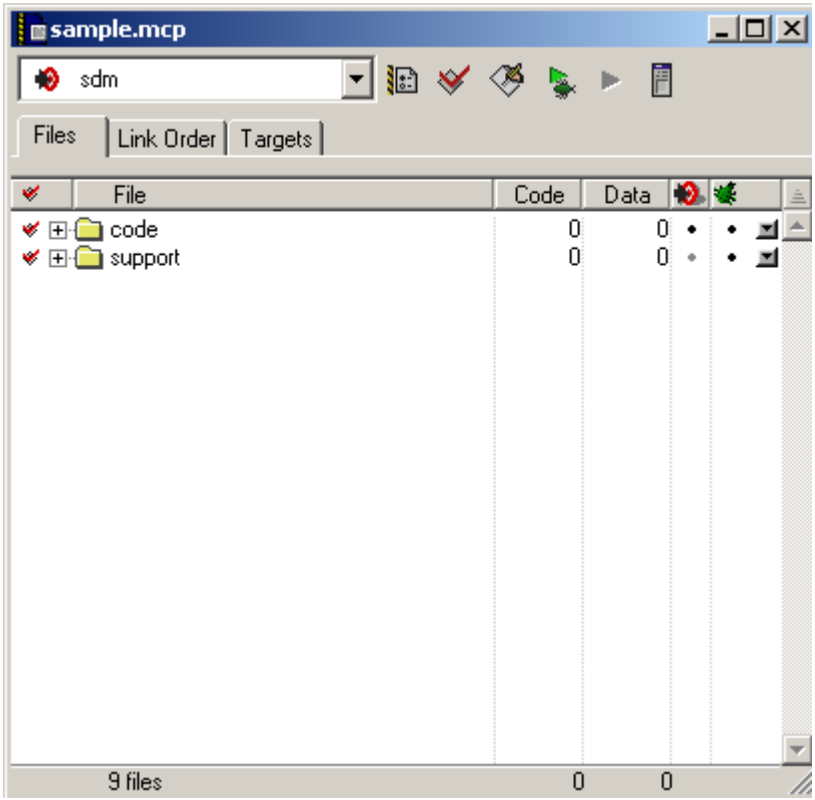
7. **Select the Project Stationery for your target.**

   In case of the Simulator, click the hierarchical control for the Project Stationery to expand the hierarchical view.Then, select “Simple C” language from the hierarchical tree.

8. **Click OK in the New Project window.**

   A project window appears (Figure 2.4). This window displays all the files and libraries that are part of the project stationery.

Figure 2.4 CodeWarrior Project Manager Window



Editing the Contents of a Project

To change the contents of a project:

1. **Add source files to the project.**

   Most stationery projects contain source files that act as placeholders. Replace these placeholders with your own files.

   To add files, use either of the following options:

   - From the menu bar of the Metrowerks CodeWarrior window, select **Project > Add Files**.
   - Drag files from the desktop or Windows Explorer to the project window.

   To remove files:

   a. **Select the files in the project window that you want to delete.**

b. Press the Delete key or right click and select remove.

2. Edit code in the source files.

Use the IDEs source-code editor to modify the content of a source-code file. To open a file for editing, use either of the following options:

- Double-click the file in the project window.
- Select the file in the project window and press **Enter**.

Once the file is open, you can use all of the editor's features to work with your code.

3

Development Studio Overview

This chapter is for new users of the CodeWarrior™ IDE. This chapter contains the following sections:

- CodeWarrior IDE
- The Development Process

If you are an experienced CodeWarrior IDE user, you will recognize the look and feel of the user interface. However, it is necessary that you become familiar with the DSP56800E runtime software environment.

CodeWarrior IDE

The CodeWarrior IDE lets you create software applications. It controls the project manager, the source-code editor, the class browser, the compiler, linker, and the debugger.

In the project manager, you can organize all the files and settings related to your project so that you can see your project at a glance and easily navigate among your source-code files. The CodeWarrior IDE automatically manages build dependencies.

A project can have multiple build targets. A build target is a separate build (with its own settings) that uses some or all of the files in the project. For example, you can have both a debug version and a release version of your software as separate build targets within the same project.

The CodeWarrior IDE has an extensible architecture that uses plug-in compilers and linkers to target various operating systems and microprocessors. The CodeWarrior CD includes a C compiler for

the DSP56800E family of processors. Other CodeWarrior software packages include C, C++, and Java compilers for Win32, Mac® OS, Linux, and other hardware and software combinations.

CodeWarrior Compiler for DSP56800E

The CodeWarrior compiler for DSP56800E is an ANSI-compliant C compiler. This compiler is based on the same compiler architecture used in all CodeWarrior C compilers. When it is used together with the CodeWarrior linker for DSP56800E, you can generate DSP56800E applications and libraries.

NOTE The CodeWarrior compiler for DSP56800E does not support C++.

CodeWarrior Assembler for DSP56800E

The CodeWarrior assembler for DSP56800E has an easy-to-use syntax. The CodeWarrior IDE assembles any file with an `.asm` extension in the project. For further information, refer to the *Assembler Reference Manual*.

CodeWarrior Linker for DSP56800E

The CodeWarrior linker for Motorola DSP56800E is in an Executable and Linker Format (ELF) linker. This linker lets you generate an ELF file (the default output file format) for your application and generate an S-record output file for your application.

CodeWarrior Debugger for DSP56800E

The CodeWarrior debugger controls your program's execution and lets you see what happens internally as your program runs. Use the debugger to find problems in your program's execution.

The debugger can execute your program one statement at a time and suspend execution when control reaches a specified point. When the debugger stops a program, you can view the chain of function calls, examine and change the values of variables, inspect the contents of the processor's registers and see the contents of memory.

Metrowerks Standard Library

The Metrowerks Standard Library (MSL) is a set of standard C libraries for use in developing DSP56800E applications. These libraries are ANSI-compliant. Access the library sources for use in your projects. These libraries are a subset of the same ones used for all platform targets, but the libraries have been customized and the runtime adapted for use in DSP56800E development.

The Development Process

While working with the CodeWarrior IDE, you proceed through the development stages familiar to all programmers: write code, compile and link code, and debug code.

The difference between the CodeWarrior IDE and traditional command-line environments is in how the software (in this case the CodeWarrior IDE) helps you manage your work more effectively. If you are not familiar with an integrated development environment in general, or with the CodeWarrior IDE in particular, you will find the topics in this section helpful.

Project Files versus Makefiles

The CodeWarrior IDE *project* is analogous to a collection of makefiles because you can have multiple builds in the same project. For example, you can have one project that maintains both a debug version and a release version of your code. You can build either or both of these versions as you wish. Different builds within a single project are called “build targets.”

The CodeWarrior IDE uses the project window to list the files in a project. A project can contain various types of files, such as source-code files and libraries.

You can easily add or remove files from a project. You can assign files to one or more build targets within the same project. These assignments let you easily manage files common to multiple build targets.

The CodeWarrior IDE automatically handles the dependencies between files, and it tracks which files have changed since the last build. When you rebuild a project, only those files that have changed are recompiled.

The CodeWarrior IDE also stores compiler and linker settings for each build target. You can modify these settings by changing the options in the target settings panels of the CodeWarrior IDE or by using `#pragma` statements in your code.

Editing Code

The CodeWarrior IDE features a text editor. It handles text files in MS-DOS®/Windows,® UNIX, and Mac® OS formats.

To open and edit a source-code file, or any other editable file in a project, use either of the following options:

- Double-click the file in the project window.
- Click the file. The file is highlighted. Drag the file to the Metrowerks CodeWarrior IDE window.

The editor window has excellent navigational features that allow you switch between related files, locate any particular function, mark any location within a file, or go to a specific line of code.

Compiling

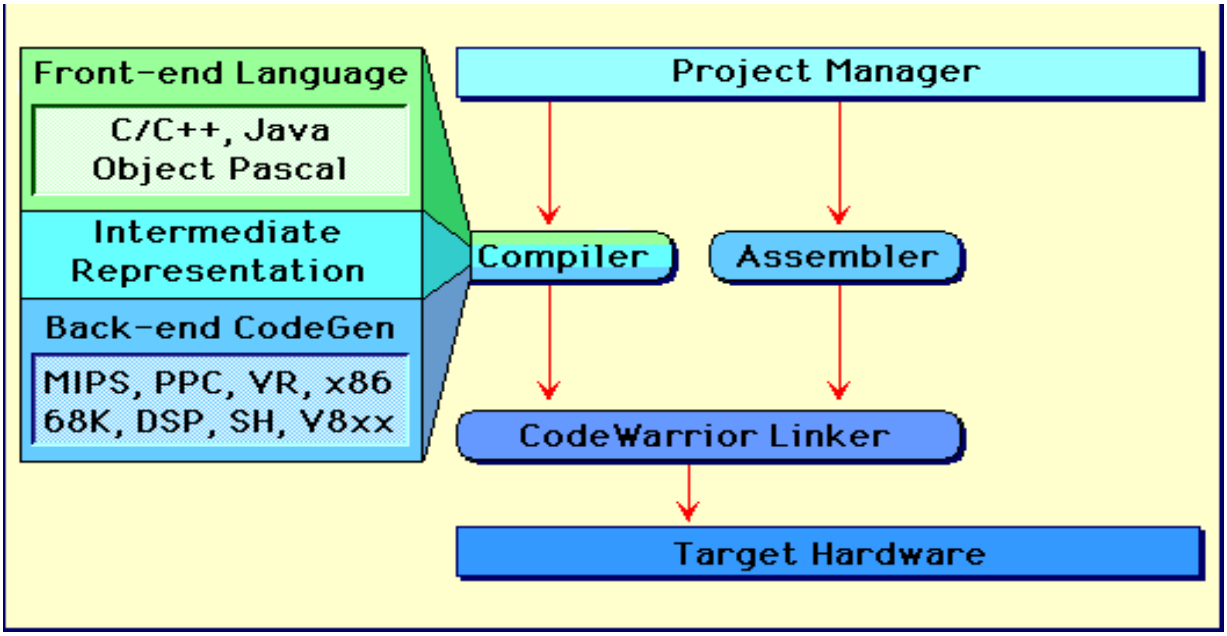
To compile any source-code file in the current build target, select the source-code file in the project window and then select **Project > Compile** from the menu bar of the Metrowerks CodeWarrior window.

To compile all the files in the current build target that were modified since they were last compiled, select **Project >Bring Up To Date** from the menu bar of the Metrowerks CodeWarrior window.

In UNIX and other command-line environments, object code compiled from a source-code file is stored in a binary file (a `.o` or `.obj` file). On Windows targets, the CodeWarrior IDE stores and manages object files internally in the data folder.

A proprietary compiler architecture is at the heart of the CodeWarrior IDE. This architecture handles multiple languages and platform targets. Front-end language compilers generate an intermediate representation (IR) of syntactically correct source code. The IR is memory-resident and language-independent. Back-end compilers generate code from the IR for specific platform targets. The CodeWarrior IDE manages the whole process (Figure 3.1).

Figure 3.1 CodeWarrior Build System



As a result of this architecture, the CodeWarrior IDE uses the same front-end compiler to support multiple back-end platform targets. In some cases, the same back-end compiler can generate code from a variety of languages. Users derive significant benefit from this architecture. For example, an advance in the C/C++ front-end compiler means an immediate advance in all code generation. Optimizations in the IR mean that any new code generator is highly optimized. Targeting a new processor does not require compiler-related changes in the source code, so porting is much more simpler.

All compilers are built as plug-in modules. The compiler and linker components are modular plug-ins. Metrowerks publishes this API, allowing developers to create custom or proprietary tools. For more information, go to Metrowerks Support at this URL:

`http://www.metrowerks.com/MW/Support`

Once the compiler generates object code, the plug-in linker generates the final executable file. Multiple linkers are available for some platform targets to support different object-code formats.

Linking

To link object code into a final binary file, select **Project > Make** from the menu bar of the Metrowerks CodeWarrior window. The **Make** command brings the active project up to date, then links the resulting object code into a final output file.

The CodeWarrior IDE controls the linker through the use of linker command files. There is no need to specify a list of object files. The Project Manager tracks all the object files automatically. You can also use the Project Manager to specify link order.

The **Target>M56800E Target** settings panel lets you set the name of the final output file.

Debugging

To debug a project, select **Project > Debug** from the menu bar of the Metrowerks CodeWarrior window.

Viewing Preprocessor Output

To view preprocessor output, select the file in the project window and select **Project > Preprocess** from the menu bar of the Metrowerks CodeWarrior window. The CodeWarrior IDE displays a new window that shows you what your file looks like after going through the preprocessor.

You can use this feature to track down bugs caused by macro expansion or other subtleties of the preprocessor.

4

Target Settings

Each build target in a CodeWarrior™ project has its own settings. This chapter explains the target settings panels for DSP56800E software development. The settings that you select affect the DSP56800E compiler, linker, assembler, and debugger.

This chapter contains the following sections:

- Target Settings Overview
- CodeWarrior IDE Target Settings Panels
- DSP56800E-Specific Target Settings Panels

Target Settings Overview

The target settings control:

- Compiler options
- Linker options
- Assembler options
- Debugger options
- Error and warning messages

When you create a project using stationery, the build targets, which are part of the stationery, already include default target settings. You can use those default target settings (if the settings are appropriate), or you can change them.

NOTE Use the DSP56800E project stationery when you create a new project.

Target Setting Panels

Table 4.1 lists the target settings panels.

Table 4.1 Target Setting Panels

Settings Group	Panel Name
Target	Target Settings
	Access Paths
	Build Extras
	Runtime Settings
	File Mappings
	Source Trees
	M56800E Target
Language Settings	C/C++ Language
	C/C++ Warnings
	M56800E Assembler
Code Generation	M56800E Processor
	Global Optimization
Linker	ELF Disassembler
	M56800E Linker
Editor	Custom Keywords
Debugger	Other Executables
	Debugger Settings
	Remote Debugging
	M56800E Target
	Remote Debug Options

Changing Target Settings

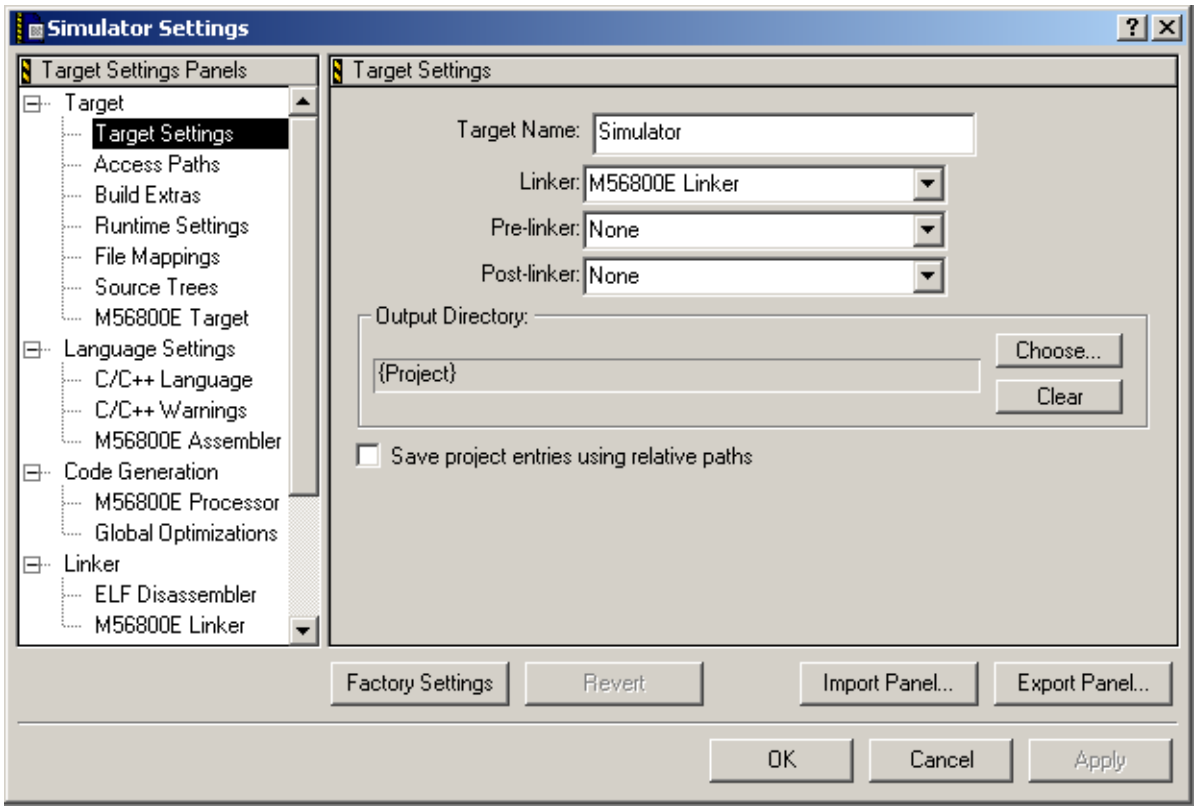
To change target settings:

1. **Select Edit > Target Name Settings.**

   Target is the name of the current build target in the CodeWarrior project.

   After you select this menu item, the CodeWarrior IDE displays the **Target Settings** window (Figure 4.1).

Figure 4.1 Target Settings Window



The left side of the **Target Settings** window contains a list of target settings panels that apply to the current build target.

2. **To view the Target Settings panel:**

   Click on the name of the **Target Settings** panel in the **Target Settings** panels list on the left side of the **Target Settings** window.

   The CodeWarrior IDE displays the target settings panel that you selected.

3. **Change the settings in the panel.**
4. **Click OK.**

Exporting and Importing Panel Options to XML Files

The CodeWarrior IDE can export options for the current settings panel to an Extensible Markup Language (XML) file or import options for the current settings panel from a previously saved XML file.

Exporting Panel Options to XML File

1. **Click the Export Panel button.**
2. **Assign a name to the XML file and save the file in the desired location.**

Importing Panel Options from XML File

1. **Click the Import Panel button.**
2. **Locate the XML file to where you saved the options for the current settings panel.**
3. **Open the file to import the options.**

Saving New Target Settings in Stationery

To create stationery files with new target settings:

1. **Create your new project from an existing stationery.**
2. **Change the target settings in your new project for any or all of the build targets in the project.**
3. **Save the new project in the Stationery folder.**

Restoring Target Settings

After you change settings in an existing project, you can restore the previous settings by using any of the following methods:

- To restore the previous settings, click **Revert** at the bottom of the **Target Settings** window.
- To restore the settings to the factory defaults, click **Factory Settings** at the bottom of the window.

CodeWarrior IDE Target Settings Panels

Table 4.2 lists and explains the CodeWarrior IDE target settings panels that can apply to DSP56800E.

Table 4.2 Code Warrior IDE Target Settings Panels

Target Settings Panels	Description
Access Paths	Use this panel to select the paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative. See *IDE User Guide*.
Build Extras	Use this panel to set options that affect the way the CodeWarrior IDE builds a project, including the use of a third-party debugger. See *IDE User Guide*.
Runtime Settings	Use this panel to set a variety of options, including: • A host application to use when debugging a non-executable file (for example, a shared library) • A working directory • Program arguments • Environment variables See *IDE User Guide*.
File Mappings	Use this panel to associate a file name extension, such as.c, with a plug-in compiler. See *IDE User Guide*.
Source Trees	Use this panel to define project-specific source trees (root paths) for use in your projects. See *IDE User Guide*.
Custom Keywords	Use this panel to change the colors that the CodeWarrior IDE uses for different types of text. See *IDE User Guide*.
Other Executables	Use this panel to specify other executables to debug while debugging the current target. See *IDE User Guide*.
Global Optimizations	Use this panel to configure how the compiler optimizes the object code. See *IDE User Guide*.
Debugger Settings	Use this panel to specify settings for the CodeWarrior debugger.

DSP56800E-Specific Target Settings Panels

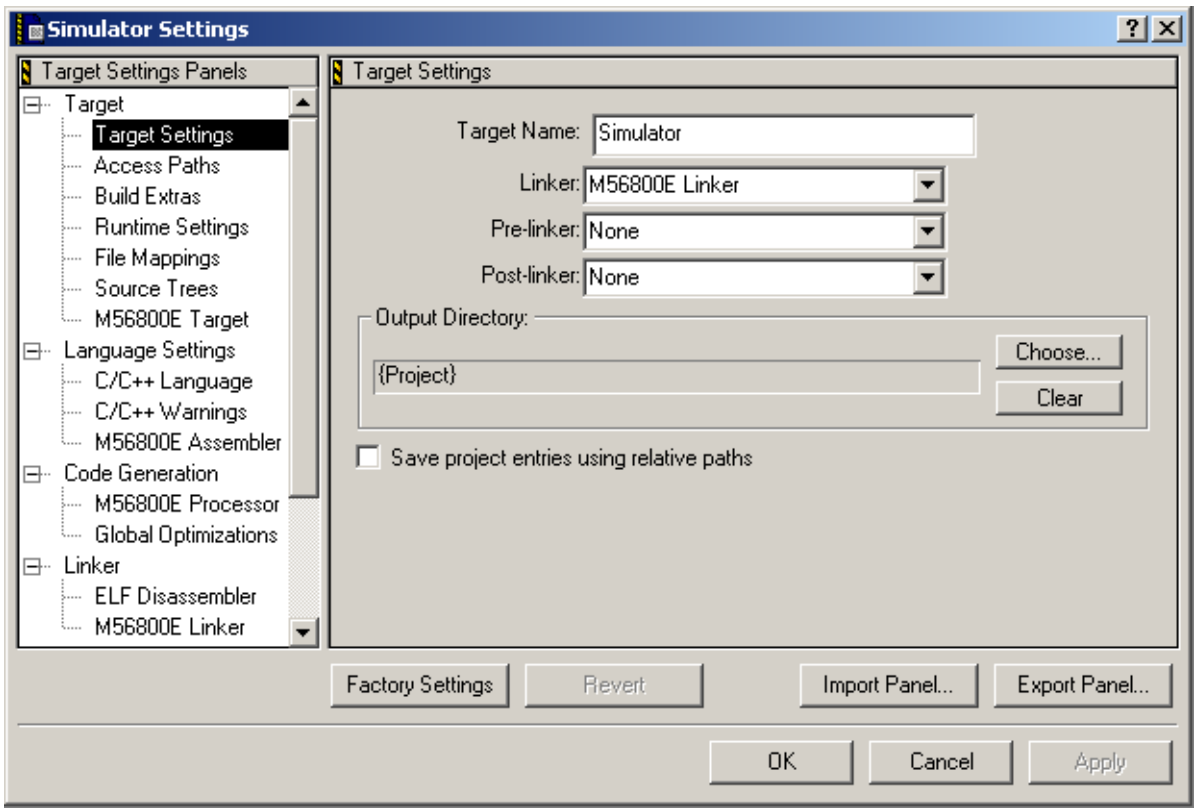
This section explains individual settings on DSP56800E-specific target settings panels.

Target Settings

The **Target Settings** panel (Figure 4.2) lets you set the name of your build target, as well as the linker and post-linker plug-ins to use with the build target. By selecting a linker, you are specifying which family of processors to use. The other available panels in the **Target Settings** window change to reflect your choice.

NOTE As the selection of the linker determines which panels are applicable to your project, first select the settings for your build target.

Figure 4.2 Target Settings Panel



The **Target Settings** panel options are:

- **Target Name**

  Use the **Target Name** field to set or change the name of a build target. When you use the **Targets** view in the project window, you see the name displayed in the **Target Name** field.

NOTE The name you specify here is not the name of your final output file. It is instead a name for your personal use that you assign to the build target. You specify the name of the final output file in the **Output File Name** field of the **Target>M56800E Target** panel.

- **Linker**

  For DSP56800E projects, you must select the M56800E Linker. After you select this linker, only the panels appropriate for your build target (in this case, DSP56800E) are available.

- **Pre-linker**

  Some build targets have pre-linkers that perform additional work (such as data-format conversion) before the final executable file is built. Because there is no pre-linker for the DSP56800E, select **None** in the **Pre-linker** list box.

- **Post-linker**

  Some build targets have post-linkers that perform additional work (such as data-format conversion) on the final executable file. Because there is no post-linker for the DSP56800E, select **None** in the **Post-linker** list box.

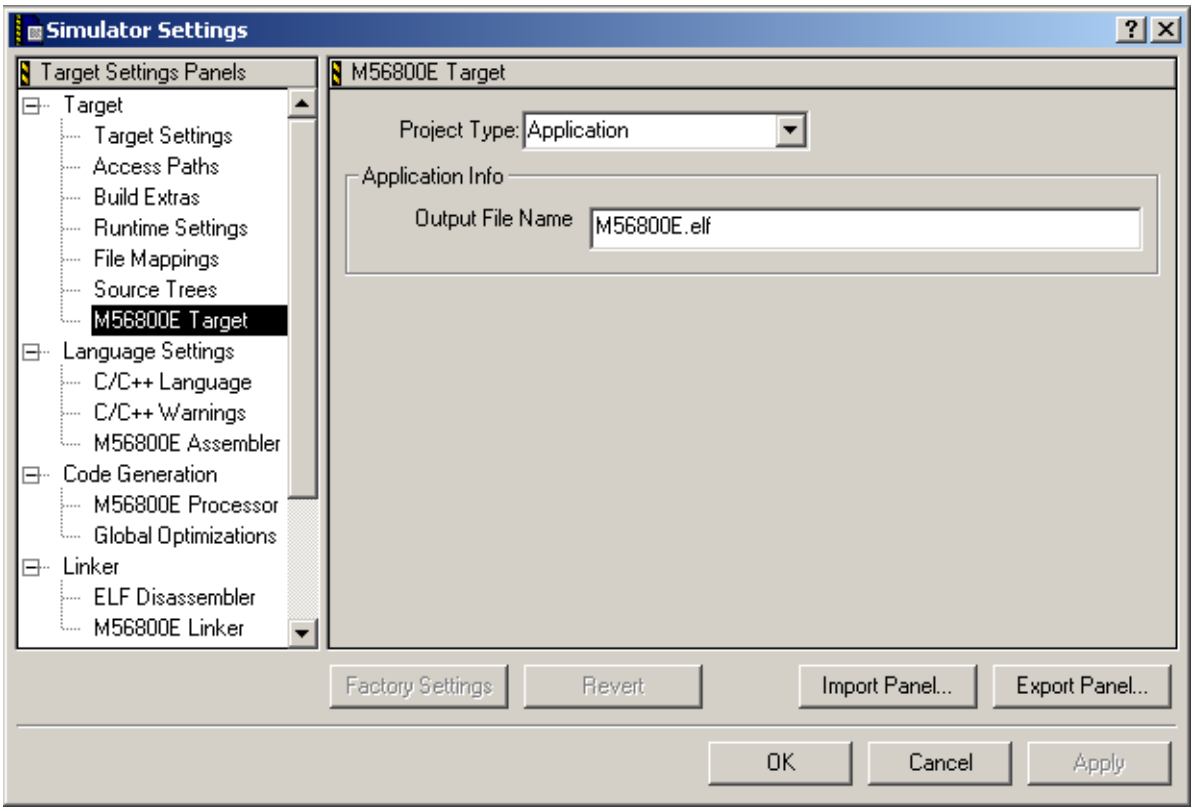
- **Output Directory**

  This field shows the directory to which the CodeWarrior IDE saves the executable file, which is built from the current project. To save the executable file to a different directory, click **Choose**. To erase the contents of this field, click **Clear**.

M56800E Target

The **M56800E Target** panel (Figure 4.3) instructs the project type and the output file name. This panel is only available when the current build target uses the M56800E Linker.

Figure 4.3 M56800E Target Panel



The **M56800E Target** panel options are:

- **Project Type**

  The **Project Type** list box determines the kind of project: **Application** and **Library**. Usually you want to build an application.

  Use this menu to select the project type that reflects the kind of project you are building.

- **Output File Name**

  The **Output File Name** field specifies the name of the executable file or library that you want to create. This file is also used by the CodeWarrior debugger. Application names end with the extension `.elf`, and library names end with the extension `.lib`.

NOTE When building a library, be sure to name it with the extension `.lib`, the default file-mapping entry for libraries. If you wish to change an extension, you must add a file-mapping entry in the **File Mappings** panel. For more information on **File Mappings** see the *IDE User Guide*.

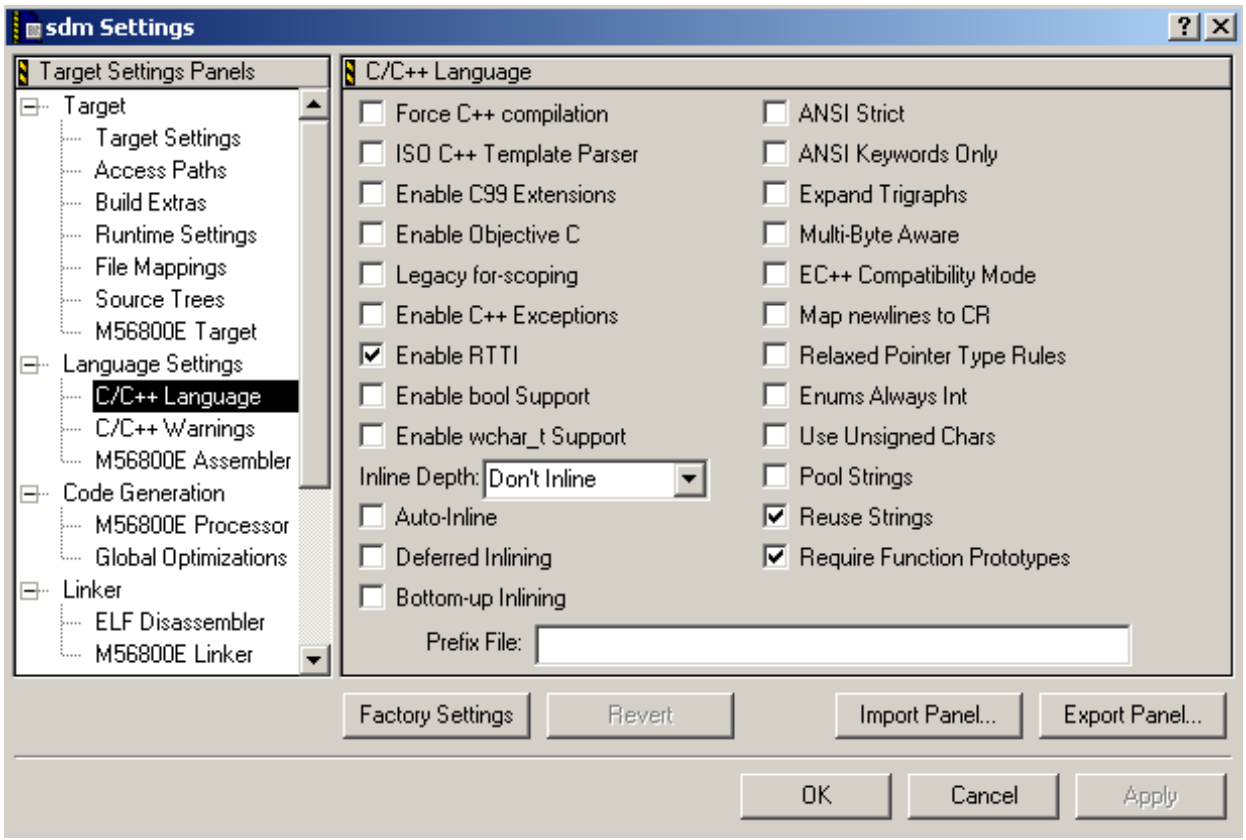
C/C++ Language

Settings in the **C/C++ Language** panel (Figure 4.4), only affect C language features implemented for the DSP56800E.

The following options are not applicable to the DSP56800E compiler. Disable the options at all times:

- Force C++ compilation
- ISO C++ Template Parser
- Enable C99 Extensions
- Enable Objective C
- Legacy for-scooping
- Enable C++ Exceptions
- Enable RTTI
- Enable bool Support
- Enable wchar_t Support
- Multi-Byte Aware
- EC++ Compatibility Mode
- Pool Strings

Figure 4.4 C/C++ Language Panel



The **C/C++ Language** panel options are:

- **Inline Depth**

  Select this function if you want the compiler to determine whether to inline a function based on the settings of **ANSI Keywords Only** and the **Inline Depth** and **Auto-inline** options.

NOTE When you call an inline function, the compiler inserts the function's code instead of issuing instructions to call that function. Inline functions makes your programs faster because you execute the function's code immediately without a function call, but possibly larger because the function's code may be repeated in several different places.

If you do not select **ANSI Keywords Only** option, you can declare C functions to be `inline`. The list box for the

Inlining options allows you to select inline no functions, only functions declared inline, or all small functions as shown in Table 4.3.

Table 4.3 Options for Inline Depth Menu

Options	Inline
Don't Inline	Does not inline functions, not even C or C++ declared `inline`.
Smart	Inline small functions to a depth of 2 to 4 inline functions deep.
1 to 8	Always inlines functions to the depth specified by the numerical selection.
Always Inline	Always inlines functions, no matter the depth.

- **Auto-Inline**

  Select this option to allow the compiler to choose which functions to inline.

  To check whether this option is on, use `__option(auto_inline)`

- **Deferred Inlining**

  Select this option if you want the compiler to allow inlining of inline and auto-inline functions that are called before these functions are declared.

  To check whether this option is on, use `__option(defer_codegen)`

  The compiler requires more memory for this option.

- **Bottom-up Inlining**

  Select this option if you want the compiler to inline functions starting at the last function to the first function in a chain of function calls.

  To check whether this option is on, use `__option(inline_bottom_up)`

- **ANSI Strict**

  This option affects several extensions to the C language supported by the CodeWarrior compiler. The extensions are:

  – C++ style comments

  – Unnamed arguments in functions definitions

- A # not followed by argument in a macro
- Using an identifier after #endif
- Using typecasted pointers as lvalues
- Converting pointers to types of the same size
- Arrays of zero length in structures
- The "D" constant suffix

In each case the extension is available only if the option is not selected. If the option is selected, then these extensions to the ANSI C standard are disabled.

To check whether this option is on, use
`__option(ANSI_Strict)`

- **ANSI Keywords Only**

  Select this option if you want the compiler to generate an error if it encounters any of the CodeWarrior C additional keywords. Use this option if you are writing code that must strictly follow the ANSI/ISO standard.

  When this option is not selected, the following additional keywords are available to you:

  - asm

    This keyword allows you to use the compiler's built-in inline assembler.

  - inline

    This keyword allows you to declare a C function as inline.

    To check whether this option is on, use
    `__option(only_std_keywords)`

- **Expand Trigraphs**

  Select this option if you want the C compiler to ignore trigraph characters. Many common character constants look like trigraph sequences (especially on Mac OS), and this extension allows you to use them without including escape characters.

  If you are writing code that must follow the ANSI/ISO standard strictly, select this option.

  To check whether this option is on, use
  `__option(trigraphs)`

NOTE If this option is on, exercise caution when you initialize strings or multi-character constants that contain questions marks.

- **Map newlines to CR**

  When you select this option, the C compiler allows you to choose how to interpret the newline ('\n') and return ('\r') characters.

  In most compilers, '\r' is translated to the value `0x0D`, the standard value for carriage return, and '\n' is translated to `0x0A`, the standard value for linefeed.

  However, the C compiler in the Macintosh Programmers Workshop, known as MPW C, '\r' is translated to `0x0A` and '\n' is translated to `0x0D` - the opposite of the typical behavior.

  If you select this option, the compiler uses the MPW C conventions for '\n' and '\r' characters.

  If you do not select this option, the compiler uses the CodeWarrior C and C++ conventions for the '\n' and '\r' characters.

  To check whether this option is on, use `__option(mpwc_newline)`

- **Relaxed Pointer Type Rules**

  When you select this option, the compiler treats `char*` and `unsigned char*` as the same type. While prototypes are checked for compatible pointer types, direct pointer assignments are allowed.

  This option is useful for if you are using code written before the ANSI/ISO standard. Old source code frequently uses these types interchangeably.

  To check whether this option is on, use `__option(mpwc_relax)`

- **Enums Always Int**

  When you select this option, the underlying type is always `signed int`. All enumerators must be no larger than a `signed int`. If an enumerated constant is larger than an `int`, the compiler generates an error.

  However, if you do not select this option, enumerators that can be represented as an `unsigned int` are implicitly converted to `signed int`. The compiler chooses the integral

data type that supports the largest enumerated constant. The type could be as small as a `char` or as large as `long int`.

To check whether this option is on, use `__option(enumalwaysint)`

- **Use Unsigned Chars**

  Select this option to allow the C compiler to treat a `char` declaration as an `unsigned char` declaration.

  To check whether this option is on, use `__option(unsigned_char)`

NOTE If you select this option, your code may not be compatible with libraries that were compiled with this option turned off.

- **Reuse Strings**

  Select this option if you want the compiler to store each string literal separately.

  When you select this option, the compiler stores only one copy of identical string literals. This option helps you save memory if your program contains identical string literals that you would not modify.

  If you select this option and if you change one of the strings, all the strings will be changed.

- **Require Function Prototypes**

  Select this option if you want the compiler to generate an error when you use a function that is defined after it is referenced and does not have a prototype. If the function is implicitly defined, that is, defined before it is referenced, and does not have a prototype, then the compiler will issue a warning when this option is on. This option helps you to prevent errors that occur when you call a function before you declare or define it. For example, without a function prototype, you may pass data of the wrong type. As a result, your code may not work as you expect even though it compiles without error.

  To check whether this option is on, use `__option(require_prototypes)`

C/C++ Warnings

Settings in the **C/C++ Warnings** panel (Figure 4.5), affect only C language features implemented for DSP56800E.

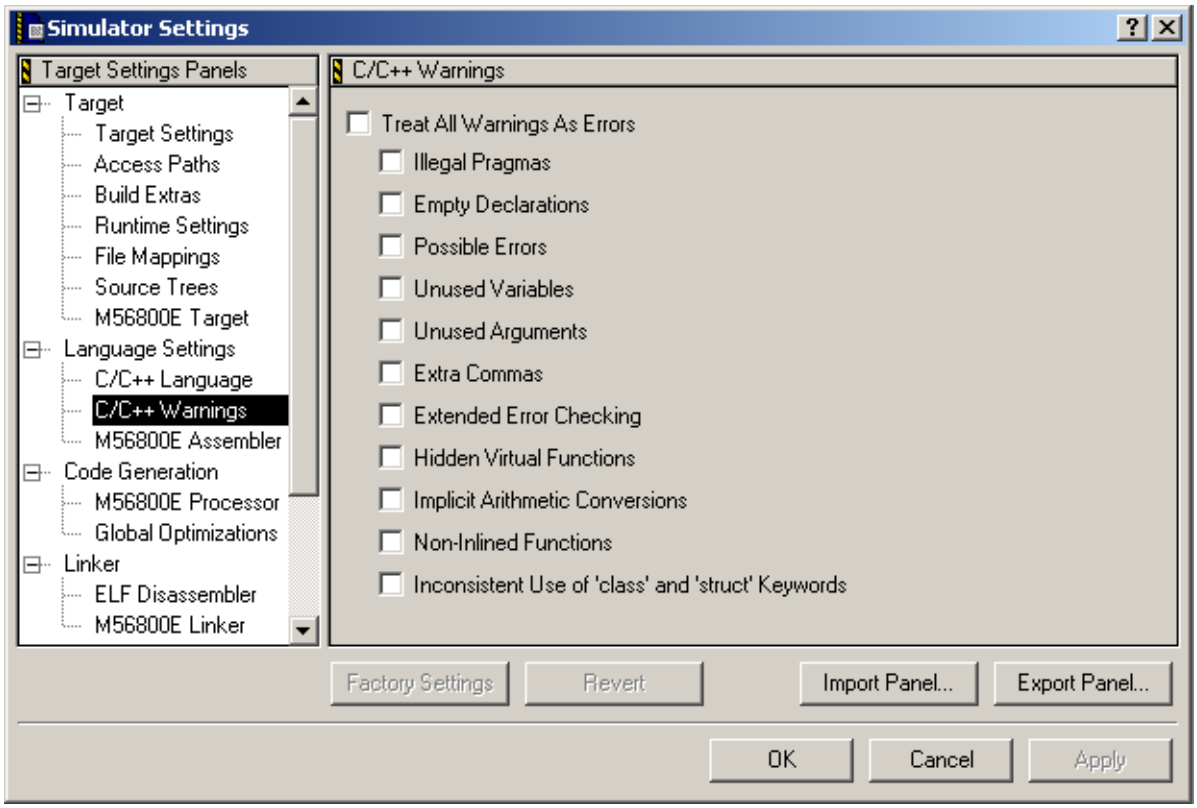
NOTE The CodeWarrior compiler for DSP56800E does not support C++.

There are no C++ warning features applicable to DSP56800E development.

The following options are not applicable to the DSP56800E compiler. Disable the options at all times:

- Hidden Virtual Functions
- Inconsistent use of 'class' and 'struct' Keywords

Figure 4.5 C/C++ Warnings Panel



The **C/C++ Warnings** panel options are:

- **Illegal Pragmas**

  When you select this option, the compiler displays a warning if it encounters an illegal pragma.

  To check whether this option is on, use `__option(warn_illpragma)`

Listing 4.1 Example of Illegal Pragma Statements that generate Warnings

```
#pragma near_data off     // WARNING: near data is not a pragma
```

- **Empty Declarations**

  When you select this option, the compiler declares a warning if it encounters a declaration with no variable name.

  To check whether this option is on, use `__option(warn_emptydecl)`

Listing 4.2 Example of Empty Declarations that generate Warnings

```
int ;         // WARNING
int i;        // OK
```

- **Possible Errors**

  Select this option if you want the compiler to check for some common typographical mistakes that are legal C syntax, but that may have unwanted side effects, such as putting in unintended semicolons or confusing = and ==. The compiler generates a warning if it encounters one of these:

  – An assignment in a logical expression or the condition in a `while`, `if`, or `for` expression. This check is useful if you use = when you meant to use ==.

  – An equal comparison in a statement that contains a single expression. This check is useful if you use == when you meant to use =.

  – A semicolon (;) directly after a `while`, `if`, or `for` statement.

  To check whether this option is on, use `__option(warn_possunwant)`

- **Unused Variables**

  When you select this option, compiler generates a warning when it encounters a variable that you declare, but do not use. This check helps you find misspelled variable names and variables you have written out of your program.

  If you want to use this warning, but need to declare a variable that you do not use, use the pragma statement "unused" on page 109.

  To check whether this option is on, use `__option(warn_unusedvar)`

- **Unused Arguments**

  When you select this option, the compiler generates a warning when it encounters an argument you declare but do not use. This check helps you find misspelled argument names and arguments you have written out of your program.

  There are two ways to avoid this warning:

  – Use the pragma `unused` statement.

  – You can turn off the **ANSI Strict** option in the **C/C++ Language Panel** and not assign a name to the unused argument.

  To check whether this option is on, use `__option(warn_unusedarg)`

- **Extra Commas**

  When you select this option, the compiler generates a warning when it encounters an extra comma in enums.

  To check whether this option is on, use `__option(warn_extracomma)`

- **Extended Error Checking**

  When you select this option, the compiler generates a warning (not an error) if it encounters one of the following syntax problems:

  – A non-void function that does not contain a `return` statement.

  – An integer or floating-point value assigned to an `enum` type.

  – An empty `return` statement (`return;`) in a function that is not declared `void`.

To check whether this option is on, use `__option(extended_errorcheck)`

- **Implicit Arithmetic Conversions**

  When you select this option, the compiler issues a warning if the destination of an operation is not large enough to hold all the possible results. For example, assigning the value of a variable type `long` to a variable of type `char` results in a warning if this option is on.

  To check whether this option is on, use `__option(warn_implicitconv)`

- **Non-Inlined Functions**

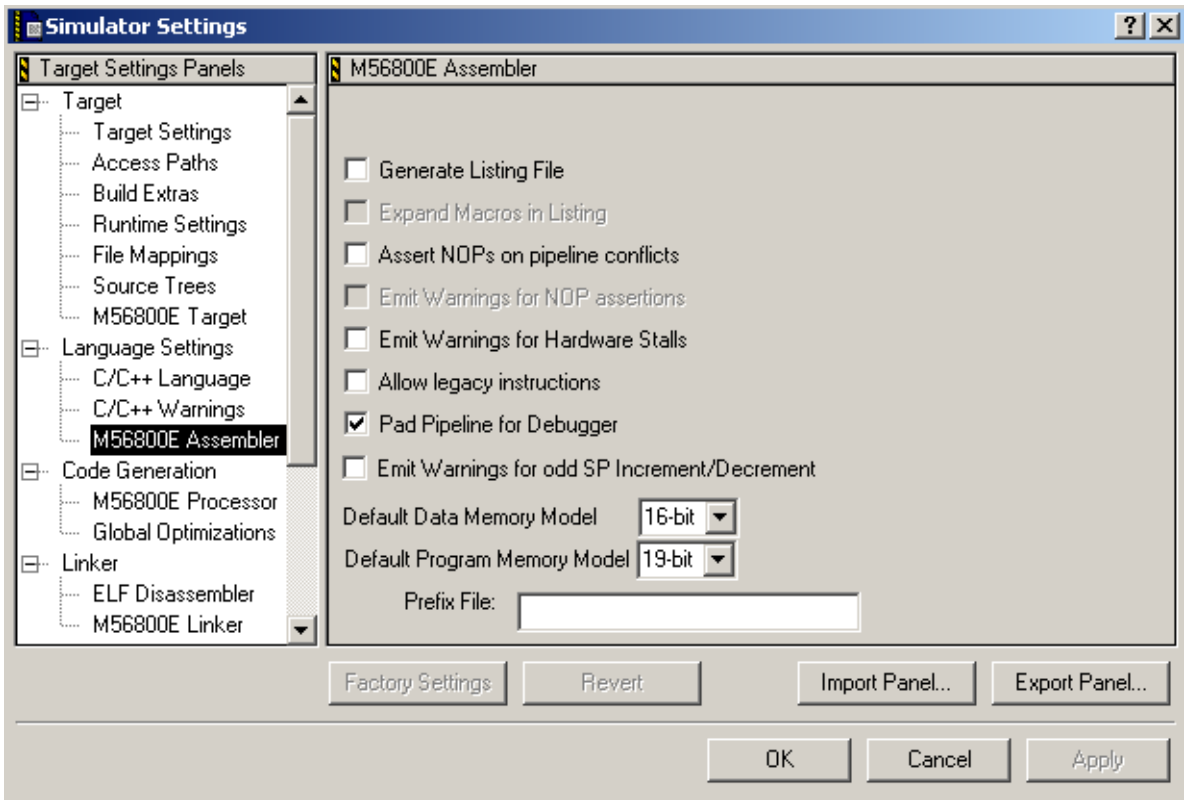
  Select this option if you want the compiler to issue a warning when it is unable to inline a function.

  To check whether this option is on, use `__option(warn_notinlined)`

M56800E Assembler

The **M56800E Assembler** panel (Figure 4.6) determines the format used for the assembly source files and the code generated by the DSP56800E assembler.

Figure 4.6 M56800E Assembler Settings Panel



The **M56800E Assembler Settings** panel options are:

- **Generate Listing File**

  The **Generate Listing File** option determines whether a listing file is generated when the CodeWarrior IDE assembles the source files of the project. If this option is enabled, the assembler creates a listing file which contains the source file along with line numbers, relocation information, and macro expansions. If this option is disabled, the assembler does not generate the listing file.

When a listing file is output, the file is created in the same directory as the assembly file and an `.lst` extension is appended to the end of the file name.

- **Expand Macros in Listing**

  This option is available only if the checkbox for **Generate Listing File** is selected.

  If this option is enabled, the assembler macros expand in the assembler listing.

- **Assert NOPs on pipeline conflicts**

  If this option is enabled, the assembler automatically inserts NOPs wherever a pipeline conflict is detected. An error will be emitted when pipeline conflicts are detected.

- **Emit Warnings for NOP Assertions**

  If this option is enabled, the assembler warns you that it inserted a NOP to avoid a pipeline conflict. This option is applicable when **Assert NOPs on pipeline conflicts** is enabled.

- **Emit Warnings for Hardware Stalls**

  If this option is enabled, the assembler warns you that a hardware stall will occur when executed. This option is used to help you optimize the cycle count.

- **Allow legacy instructions**

  If this option is enabled, the assembler allows the legacy DSP56800 instruction syntax and sets the **Default Data Memory Model** and the **Default Program Memory Model** to 16 bits.

- **Pad Pipeline for Debugger**

  You must enable this option when using the debugger. This option inserts a NOP after certain branch instructions required to make the breakpoints work reliably. Due to the extra NOP instructions, there is a growth of about 5% in the code. If you do not set this option, you may not recover the branch instructions after the debugger comes across a breakpoint. Also, select the **Pad pipeline for debugger** option in the **M56800E Processor Settings** panel (Figure 4.8).

- **Emit Warnings for odd SP Increment/Decrement**

  This enables assembler warnings on instructions that can misalign the stackframe.

- **Default Data Memory Model**

The factory setting for this memory is 16 bits; however, you can set this memory to 24 bits.

- **Default Program Memory Model**

  The factory setting for this memory is 19 bits; however, you can set this memory to 16 or 21 bits.

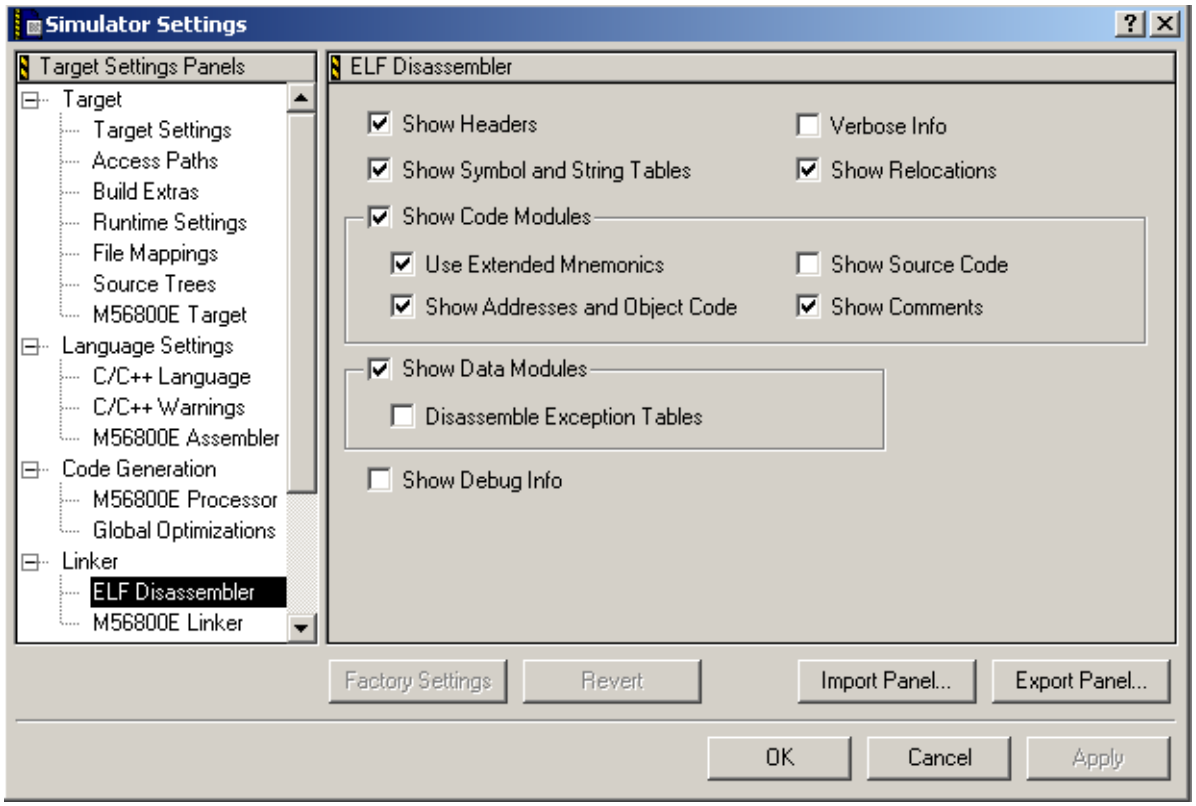
- **Prefix File**

  The **Prefix File** specifies a file to be included at the beginning of every assembly file in the project. This field lets you include common definitions without using an include directive in every file.

ELF Disassembler

The **ELF Disassembler** panel (Figure 4.7) appears when you disassemble object files. To view the disassembly of a module, select **Project > Disassemble**.

Figure 4.7 ELF Disassembler Panel



The **ELF Disassembler** panel options are:

- **Show Headers**

  The **Show Headers** option determines whether the assembled file lists any ELF header information in the disassembled output.

- **Show Symbol and String Tables**

  The **Show Symbol and String Tables** option determines whether the disassembler lists the symbol and string table for the disassembled module.

- **Verbose Info**

  The **Verbose Info** option instructs the disassembler to show additional information in the ELF file. For the `.symtab` section, some of the descriptive constants are shown with their numeric equivalents. The sections `.line` and `.debug` are shown with an unstructured hex dump.

- **Show Relocations**

  The **Show Relocations** option shows relocation information for the corresponding text (`.rela.text`) or data (`.rela.data`) section.

- **Show Code Modules**

  The **Show Code Modules** option determines whether the disassembler outputs the ELF code sections for the disassembled module.

  If enabled, the **Use Extended Mnemonics, Show Source Code, Show Addresses and Object Code,** and **Show Comments** options become available.

  – **Use Extended Mnemonics**

    The **Use Extended Mnemonics** option determines whether the disassembler lists the extended mnemonics for each instruction of the disassembled module.

    This option is available only if the **Show Code Modules** option is enabled.

  – **Show Addresses and Object Code**

    The **Show Addresses** and **Object Code** option determines whether the disassembler lists the address and object code for the disassembled module.

    This option is available only if the **Show Code Modules** option is enabled.

  – **Show Source Code**

    The **Show Source Code** option determines whether the disassembler lists the source code for the current module. Source code is displayed in mixed mode with line number information from the original C source.

    This option is available only if the **Show Code Modules** option is enabled.

– **Show Comments**

The **Show Comments** option displays comments produced by the disassembler, in sections where comment columns are provided.

This option is available only if the **Show Code Modules** option is enabled.

- **Show Data Modules**

The **Show Data Modules** option determines whether or not the disassembler outputs any ELF data sections (such as `.data` and `.bss`) for the disassembled module.

– **Disassemble Exception Tables**

The **Disassemble Exception Tables** option determines whether or not the disassembler outputs any C++ exception tables for the disassembled module.

This option is available when you select **Show Data Modules**.

NOTE **Disassemble Exception Tables** is not available for DSP56800E, since it does not support C++.

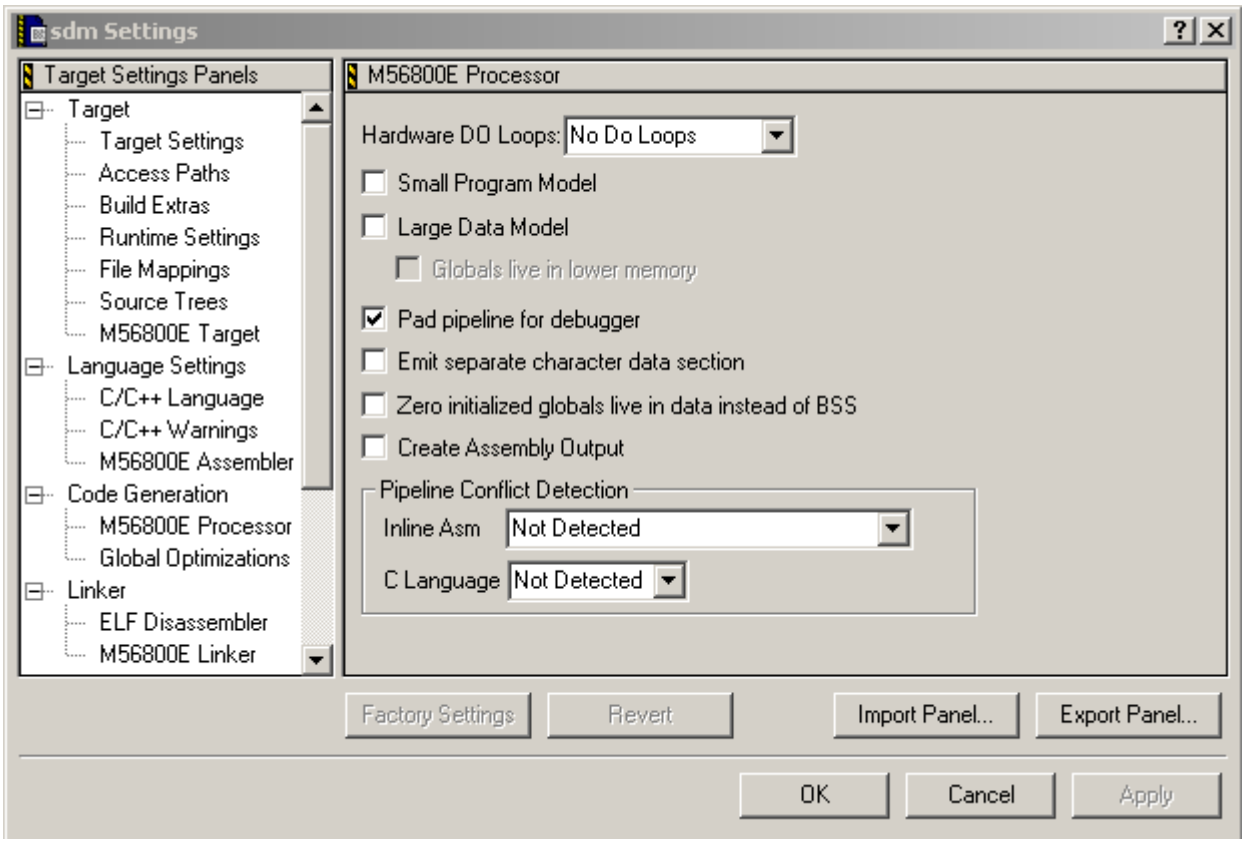
- **Show Debug Info**

The **Show Debug Info** option directs the disassembler to include DWARF symbol information in the disassembled output.

M56800E Processor

The **M56800E Processor** settings panel (Figure 4.8) determines the kind of code the compiler creates. This panel is available only if the current build target uses the M56800E Linker.

Figure 4.8 M56800E Processor Panel



The **M56800E Processor** panel options are:

- **Hardware DO Loops**

  This option controls compiler support for the different levels of hardware DO loops. From the list box, select any of the following options to specify the level of nested loops that the compiler may generate:

  – **No Do Loops –** compiler will not generate any DO loops
  – **No Nest DO Loops –** compiler can generate DO loops, but will not generate nested DO loops
  – **Nested DO Loops –** compiler can generate DO loops that are nested up to two deep (i.e., the outer and inner loops can both be DO loops)

NOTE An "inner loop" is nested inside an "outer loop" if it is entirely contained in it.

The compiler generates hardware DO loops for:

1. **aggregate (array and structure) initializations and for struct copy under the following conditions:**
   - aggregate is byte aligned and aggregate size is greater than four bytes; or
   - aggregate is word aligned, and aggregate size is greater than four words; or
   - aggregate is long aligned and
     – if **Optimize for Smaller Code Size** is selected in the **Global Optimizations** panel (as described in the *IDE User's Guide*) , aggregate size is greater than eight words.
     – if **Optimize for Faster Execution Speed** is selected, aggregate size is greater than 32 words.
2. **counted loops in C if the loop counter is less than 65536 and there are no jumps to subroutines inside the loop.**

- **Small Program Model**

  If enabled, the **Small Program Model** option allows for more efficient switch table generation. Enable this option only if the entire program code fits in 0x0-0xFFFF.

- **Large Data Model**

  If enabled, extends the data addressing range of the DSP56800E by providing 24-bit address capability to some

instructions. 24-bit address modes allows access beyond the 64K word boundary set by 16-bit addressing.

– **Globals live in lower memory**

This option is available only if the checkbox for Large Data Model is selected. This option instructs the compiler to access global and static data with 16-bit addresses while all pointer and stack operations are performed using 24-bit addressing.

- **Pad pipeline for debugger**

  You must enable this option when using the debugger. This option inserts a NOP after certain branch instructions required to make the breakpoints work reliably. Due to the extra NOP instructions, there is a growth of about 5% in the code. If you do not set this option, you may not recover the branch instructions after the debugger comes across a breakpoint. Also, select the **Pad pipeline for debugger** option in the **M56800E Assembler Settings** panel (Figure 4.6).

- **Emit separate character data section**

  If enabled, all character data is broken out and placed into its own data sections.

  The compiler automatically selects the proper section. The available sections are:

.data.char	Initialized static or global character data objects are placed here.
.bss.char	Uninitialized static or global character data objects are placed here.
.const.data.char	Const qualified character objects and static string data are placed here.

  You can locate these sections in the lower half of the memory map, thus ensuring that the data can be addressed. For more information, see "Additional Information about Character Data Addressing" on page 90.

  Additionally, structures containing character data will be placed in character data sections.

- **Zero initialized globals live in data instead of BSS**

  If enabled, then the globals that are initialized to zero reside in the `.data` section instead of the `.bss` section.

- **Create Assembly Output**

  If enabled, then every `.c` file will have assembly source code generated. The pragma `#pragma asmoutput` can be used to turn off individual files. For more details see "asmoutput" on page 97.

- **Pipeline Conflict Detection**

  Use this option to detect and generate a warning or error message if the inline assembly source code and C language source code have pipeline conflicts when they are compiled. In some cases, the source code can be a mix of assembly language and C language.

  – **Inline Asm**

    This option is for detecting pipeline conflicts in the Inline Assembly source code.

    From this list box, select any of the following options:

    - Not Detected

    - Conflict Error

    - Conflict Error/Hardware Stall Warning

  – **C Language**

    This option is for detecting pipeline conflicts in the C source code.

    From this list box, select any of the following options:

    - Not Detected

    - Conflict Error

  For more details on how to use pragmas to detect pipeline conflicts, see the following:

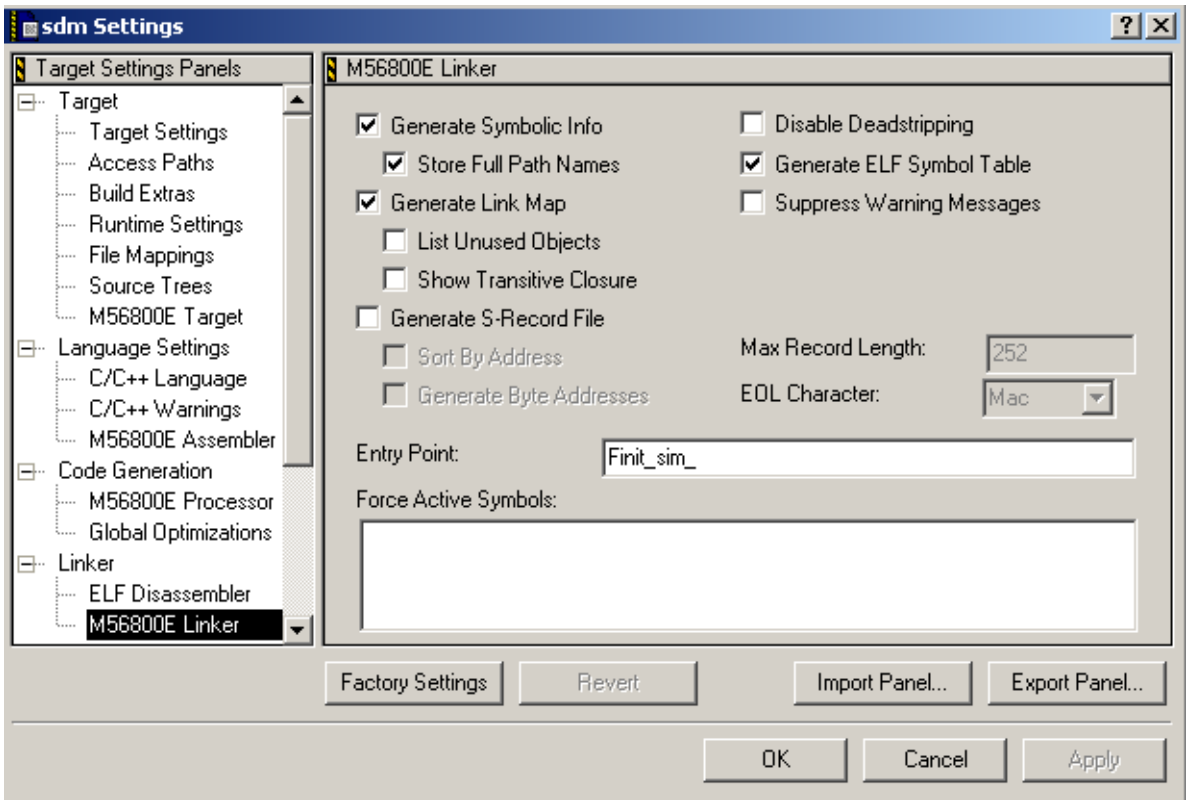
  – `#pragma check_c_src_pipeline` on page 98

  – `#pragma check_inline_asm_pipeline` on page 98

M56800E Linker

The **M56800E Linker** panel (Figure 4.9) controls the behavior of the linker. This panel is only available if the current build target uses the M56800E Linker.

Figure 4.9 M56800E Linker Panel



The **M56800E Linker** panel options are:

- **Generate Symbolic Info**

  The **Generate Symbolic Info** option controls whether the linker generates debugging information. If the option is enabled, the linker generates debugging information. This information is included within the linked ELF file. This setting does not generate a separate file.

  If you select **Project > Debug**, the CodeWarrior IDE enables the **Generate Symbolic Info** option for you.

If the **Generate Symbolic Info** option is not enabled, the **Store Full Path Names** option is not available.

NOTE If you decide to disable the **Generate Symbolic Info** option, you cannot debug your project using the CodeWarrior debugger. For this reason, the compiler enables this option by default.

– **Store Full Path Names**

The **Store Full Path Names** option controls how the linker includes path information for source files when generating debugging information.

If this option is enabled, the linker includes full path names to the source files. If this option is disabled, the linker uses only the file names. By default, this option is enabled.

This option is available only if you enable the Generate Symbolic Info option.

- **Generate Link Map**

The **Generate Link Map** option controls whether the linker generates a link map. Enable this option to let the linker generate a link map.

The file name for the link map adds the extension `.xMAP` to the generated file name. The linker places the link map in the same folder as the output `.elf` file.

For each object and function in the output file, the link map shows which file provided the definition. The link map also shows the address given to each object and function, a memory map of where each section resides in memory, and the value of each linker-generated symbol.

Although the linker aggressively strips unused code and data when the CodeWarrior IDE compiles the relocatable file, it never deadstrips assembler relocatable files or relocatable files built with other compilers. If a relocatable file was not built with the CodeWarrior C compiler, the link map lists all of the unused but unstripped symbols.

– **List Unused Objects**

The **List Unused Objects** option controls whether the linker includes unused objects in the link map. Enable the option to let the linker include unused objects in the link

map. The linker does not link unused code in the program.

Usually, this option is disabled. However, you might want to enable it in certain cases. For example, you might discover that an object you expect to be used is not actually used. This option is not available unless you enable the **Generate Link Map** option.

– **Show Transitive Closure**

The **Show Transitive Closure** option recursively lists in the link map file all of the objects referenced by main( ). Listing 4.3 shows some sample code. To show the effect of the **Show Transitive Closure** option, you must compile the code.

Listing 4.3 Sample Code to Show Transitive Closure

```
void foot( void ){  int a = 100; }
void pad( void ){ int b = 101; }

int main( void ){
  foot();
  pad();
  return 1;
}
```

After you compile the source, the linker generates a link map file. Note that this option is not available unless you enable the **Generate Link Map** option.

Listing 4.4 Effects of Show Transitive Closure in the Link Map File

```
# Link map of Finit_sim_
  1] interrupt_vectors.text found in 56800E_vector.asm
   2] sim_intRoutine (notype,local) found in 56800E_vector.asm
   2] Finit_sim_ (func,global) found in 56800E_init.asm
    3] Fmain (func,global) found in M56800E_main.c
     4] Ffoot (func,global) found in M56800E_main.c
     4] Fpad (func,global) found in M56800E_main.c
    3] F__init_sections (func,global) found in Runtime 56800E.lib
initsections.o
     4] Fmemset (func,global) found in MSL C 56800E.lib mem.o
      5] F__fill_mem (func,global) found in MSL C 56800E.lib
mem_funcs.o
  1] Finit_sim_ (func,global) found in 56800E_init.asm
```

- **Disable Deadstripping**

  The **Disable Deadstripping** option prevents the linker from removing unused code and data.

- **Generate ELF Symbol Table**

  The **Generate ELF Symbol Table** option instructs the linker to generate an ELF symbol table, as well as a list of relocations in the ELF executable file.

- **Suppress Warning Messages**

  The **Suppress Warning Messages** option controls whether the linker displays warnings. If this option is disabled, the linker displays warnings in the Message window. If this option is disabled, the linker does not display warnings.

- **Generate S-Record File**

  The **Generate S-Record File** option controls whether the linker generates an S-record file based on the application object image. The S-record files have the extension `.s`.

  In the case of the DSP56800E, the linker generates three different S-record files. Their contents are:

  - `{output file name}.s`

    S-record file containing both P and X memory contents.

  - `{output file name}.p`

    S-record file containing P memory contents only.

– `{output file name}.x`

S-record file containing X memory contents only.

The linker places the S-record files in the output folder, which is a sub-folder of the project folder.

The linker generates the following S3 type S-records:

– **Sort by Address**

This option enables the compiler to sort S-records generated by the linker using byte address. This option is not available unless you enable the Generate S-Record File option.

– **Generate Byte Addresses**

This option enables the linker to generate S-records in bytes. This option is not available unless you enable the Generate S-Record File option.

– **Max Record Length**

The **Max Record Length** field specifies the maximum length of the S-record generated by the linker. This field is available only if the Generate S-Record File option is enabled. The maximum value for an S-record length is 256 bytes.

NOTE Most programs that load applications onto embedded systems have a maximum length for S-records. The CodeWarrior debugger can handle S-records as large as 256 bytes. If you are using something other than the CodeWarrior debugger to load your embedded application, you need to determine its maximum length.

– **EOL Character**

The **EOL Character** list box defines the end-of-line character for the S-record file. This list box is available only if you enable the Generate S-Record File option.

- **Entry Point**

  The starting point for a program is set in the **Entry Point** field in the M56800E Linker settings panel. The **Entry Point** field specifies the function that the linker first uses when the program runs.

  The default function found in this field is located within the startup code that sets up the DSP56800E environment before your code executes. This function and its corresponding startup code will be different depending upon which stationery you have selected. In the case of hardware targeted stationery, the startup code can be found in the following path:

  `support\<name of hardware, e.g., M56852E>\startup`

  In the case of simulator targeted stationery, the startup code can be found in the following path:

  `support\M56800E\init`

  The `startup code` performs other tasks, such as clearing the hardware stack, creating an interrupt table, and getting the stack start and exception handler addresses.

  The final task performed by the startup code is to call your `main()` function.

- **Force Active Symbols**

  The **Force Active Symbols** field instructs the linker to include symbols in the link even if the symbols are not referenced. In essence, it is a way to make symbols immune to deadstripping. When listing multiple symbols, use a single space between them as a separator.

Remote Debugging

The **Remote Debugging** panel (Figure 4.10 or Figure 4.11) and the **M56800E Target** panel (Figure 4.13) let you set communication connections for interaction between a DSP56800E board or Simulator and the CodeWarrior DSP56800E debugger.

The **Remote Debugging** panel options are:

- **Connection Settings**

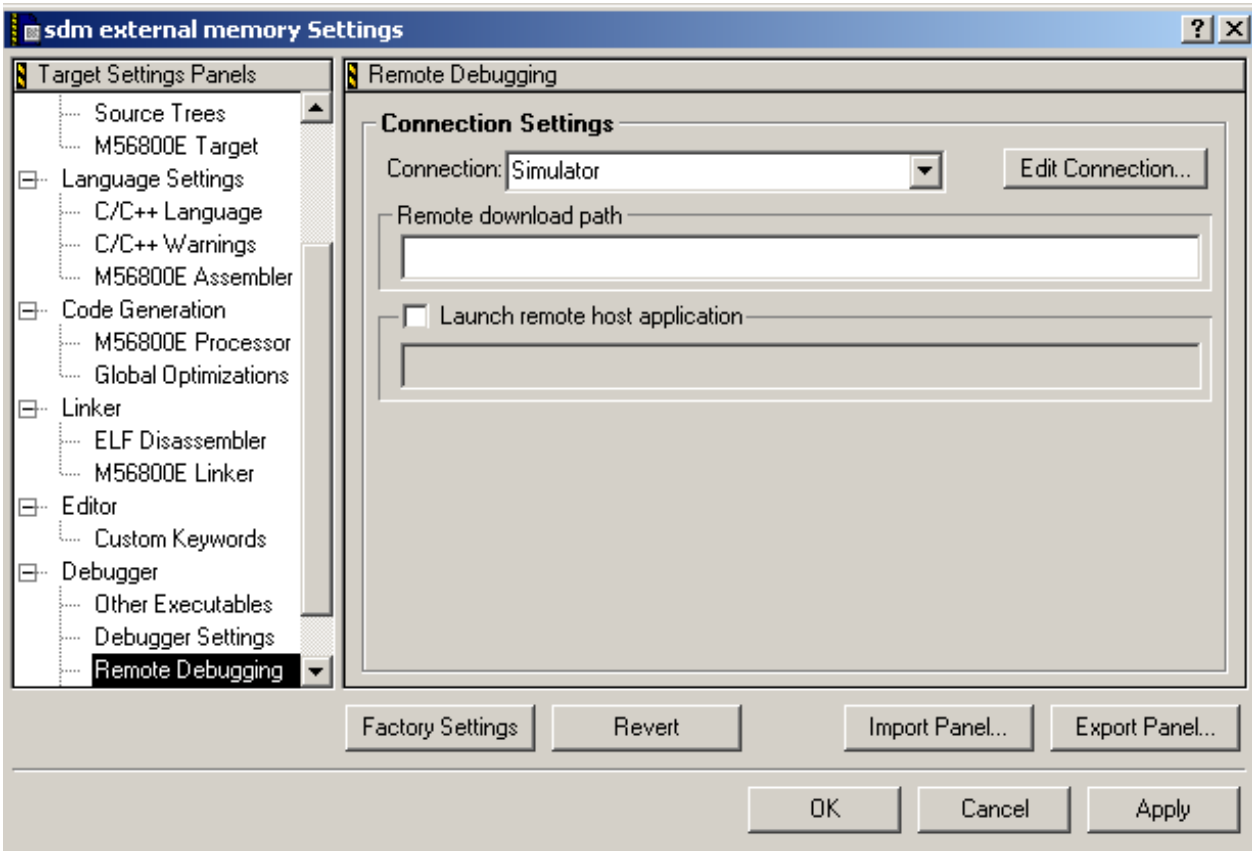
  The **Connection** list box allows you to choose the previously defined remote connection that you want to use for debugging.

For example, you may use a previously defined connection based on the simulator remote connection type called "Simulator," or you may choose to use a connection based on the CCS (command converter server) called "Local Hardware Connection (CCS)."

– **Simulator**

Select the "Simulator" in the **Connection** list box, to use the DSP56800E Simulator to test your code instead of downloading the code to actual DSP56800E hardware. The **Remote Debugging** panel shown in Figure 4.10 appears.

Figure 4.10 Remote Debugging with the Simulator

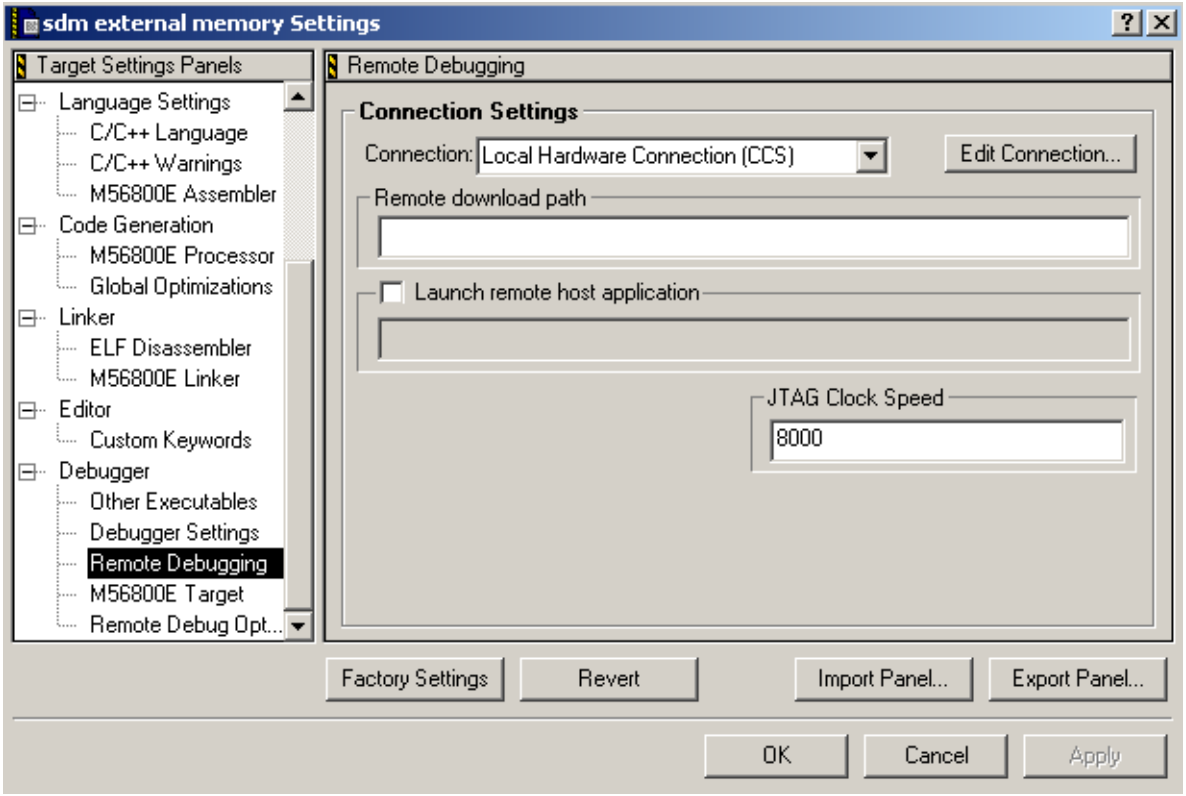


– **Command Converter Server**

Select the "Local Hardware Connection (CCS)" in the **Connection** list box, if you want use the command converter server with your computer connected to a

DSP56800E board. The **Remote Debugging** panel is shown in Figure 4.11.

Figure 4.11 Remote Debugging for the Command Converter Server (Local Connection)



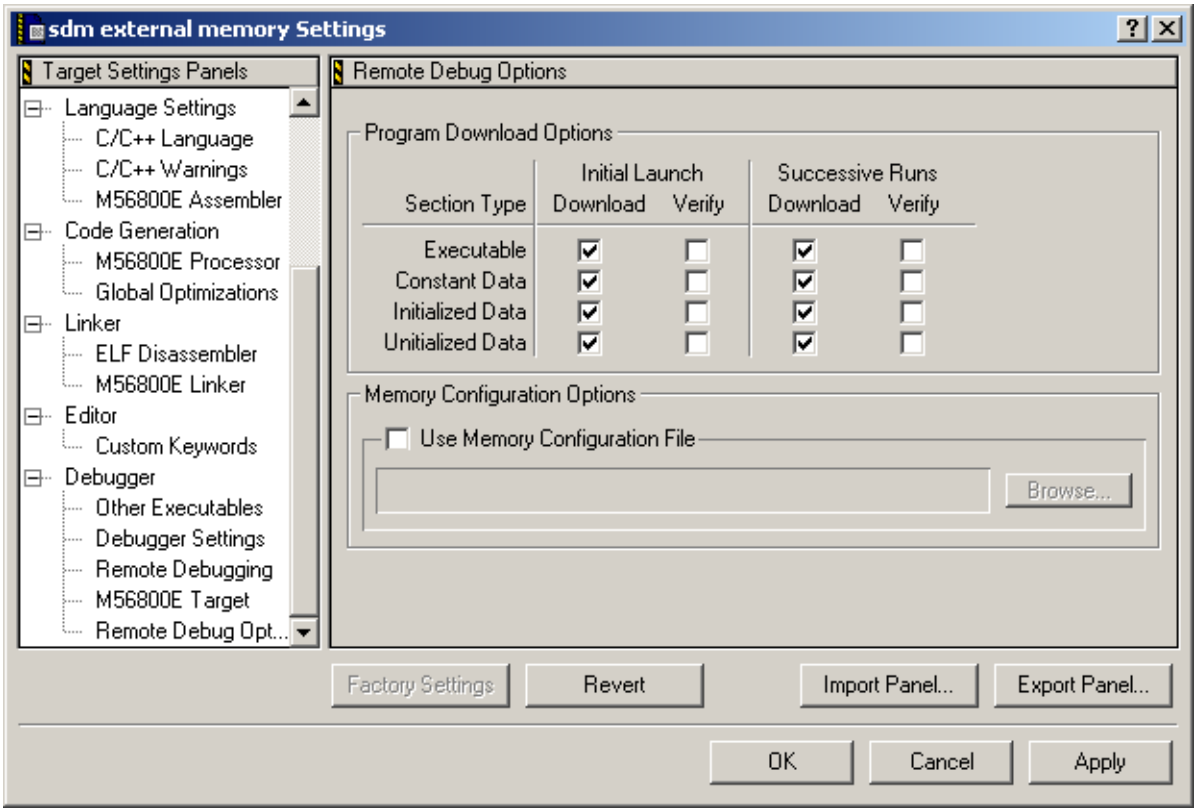
If you select the **Command Converter Server**, then the option **JTAG Clock Speed** appears with a default value of 8000 kHz.

- **Remote download path** – this option is not supported at this time
- **Launch remote host application** – this option is not supported at this time

Remote Debug Options

The **Remote Debug Options** panel (Figure 4.12) lets you select different remote debug options.

Figure 4.12 Remote Debug Options



The **Remote Debug Options** panel are:

- **Program Download Options**

  The code types are determined by the flags (RWX) assigned to a section as defined in the linker command file.

  The section types are:

  – **Executable**

    Program code that is contained in sections specified with an X flag.

  – **Constant Data**

    Program data that is contained in sections in which the X and W flags are not specified.

  – **Initialized Data**

Program data that is contained in sections in which the W flag is specified and the X flag is not specified. It is data that has been set with initial values.

– **Uninitialized Data**

Program data that is contained in sections in which the W flag is specified and the X flag is not specified. It is data that has not been set with initial values.

For the different section types you can enable **Download** or **Verify** on **Initial Launch** and on **Successive Runs**.

– **Initial Launch**

If **Download** is checked, the corresponding section is downloaded on the initial launch of the debugger.

If **Verify** is also checked, the data written is read back to verify that the data was corectly written.

– **Successive Runs**

The **Download** and **Verify** checkboxes in **Successive Runs** have the same meaning as those in the **Initial Launch** part of the table, exept that they apply to those launches after the initial launch. This can sometimes be useful in decreasing the download time for successive runs, when it is known that the memory of a given section has not changed from the previous debug session.

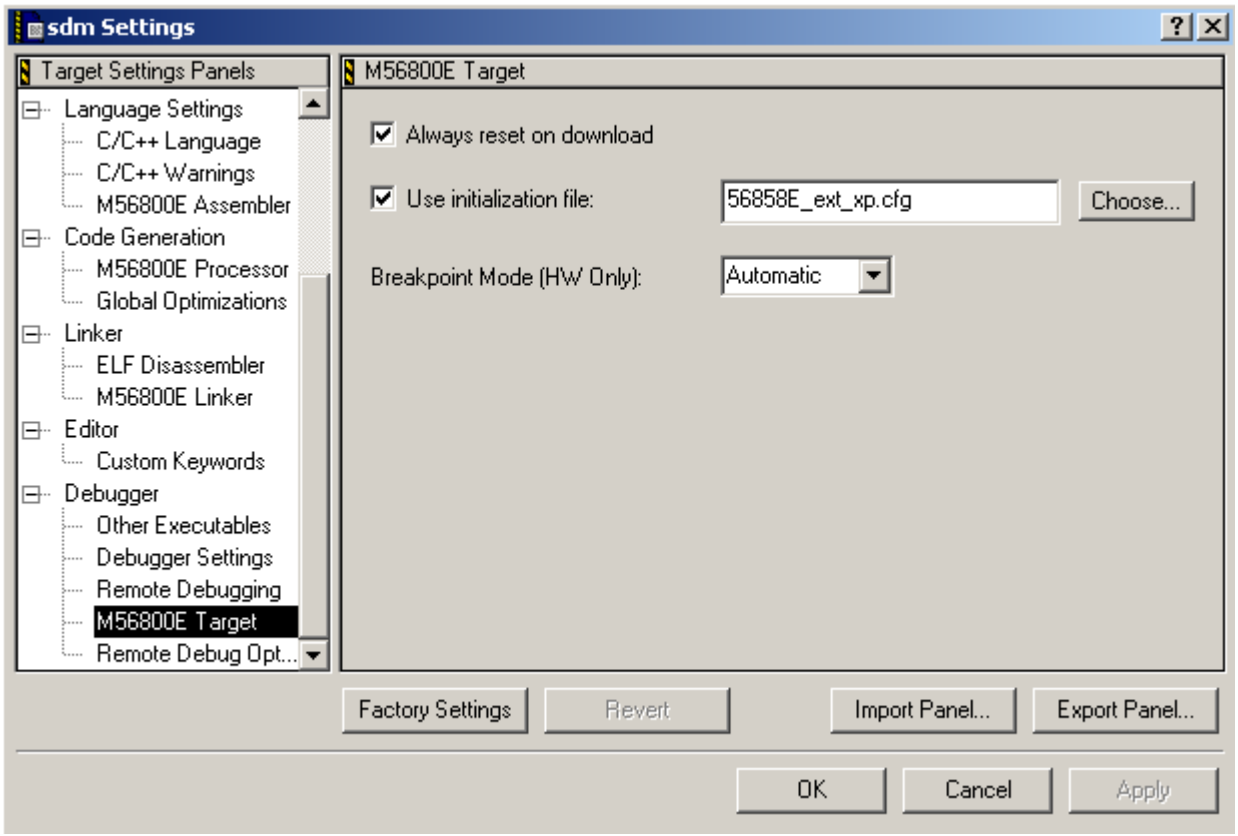
- **Memory Configuration Options**
  – **Use Memory Configuration File**

    This option is not used by the DSP56800E debugger.

M56800E Target

The **M56800E Target** panel (Figure 4.13) and the **Remote Debugging** panel (Figure 4.10 or Figure 4.11) let you set communication protocols for interaction between a DSP56800E board or Simulator and the CodeWarrior DSP56800E debugger.

Figure 4.13 M56800E Target Panel



The **M56800E Target** panel options are:

- **Always reset on download**

  Select this checkbox to cause the CodeWarrior IDE to issue a reset to the target board each time you connect to the board.

- **Use initialization file**

  Optionally, you may want to specify an initialization file. The initialization file is a text file that instructs the debugger how to initialize the hardware after reset, but before downloading code. Use the initialization file commands to write values to various registers, memory locations, and to set-up flash memory parameters.

  Four initialization files are provided: 56852E_ext_xp.cfg, 56858E_ext_xp.cfg, 56838E_ext_xp.cfg, and 56838E_Flash.cfg. They are located at the following path:

  ```
  {CodeWarrior path}\M56800E Support\
  initialization
  ```

  The first three files enable external memory on the DSP56852, DSP56858 and DSP56838 respectively. The file 56838E_Flash.cfg enables flash memory on the DSP56838. Use the appropriate file if you intend to use external memory or program the flash memory.

  To set up the initialization file:

1. **Select the checkbox Use initialization file in the M56800E Target panel.**
2. **Use either of the following options to enter the file name:**

   - Type in the file name in the text box. If a full path is not specified, the debugger searches for the file in the project directory. If the file is not found, the debugger then searches for the file in the following path:

     ```
     {CodeWarrior path}\M56800E Support\
     initialization directory.
     ```

   - Click the **Choose** button in the panel. The **Choose file** dialog box appears. Navigate to the file you want to select. The selected file name appears in the text box.

     Each line in the file begins with a command or the "#" character indicating that the line is ignored. Blanks lines are also ignored.

     Table 4.4 lists the supported commands and their arguments. For a more detailed description of the Flash

Memory commands see "Flash Memory Commands" on page 246.

Table 4.4 Commands and Arguments for Initialization Files

Commands	Argument	Description
`writepmem`	<addr> <value>	Writes a 16-bit value to the specified P: Memory location.
`writexmem`	<addr> <value>	Writes a 16-bit value to the specified X: Memory location.
`writereg`	<regName> <value>	Writes a 16-bit value to the specified register.
`set_hfmclkd`	<value>	Writes the value which represents the flash memory's clock divider to the hfmclkd register
`set_hfm_base`	<address>	Sets the address of hfm_base, which is where the flash memory control registers are mapped in X: Memory.
`add_hfm_unit`	<startAddr><endAddr> <bank><numSectors> <pageSize><progMem> <boot><interleaved>	Adds a flash memory unit to the list and sets its parameters.
`set_hfm_erase_mode`	<units | pages | all>	Sets the erase mode.
`set_hfm_verify_erase`	<1 | 0>	Set the flash memory erase verification mode.
`set_hfm_verify_program`	<1 | 0>	Sets the flash program verification mode.

NOTE The `writepmem`, `writexmem` and `writereg` commands are executed in order after the target is reset and before debugging begins.

- **Breakpoint Mode [HW Only]**

From this list box choose one of the following or accept the default, which is **Automatic**:

- **Automatic** – the CodeWarrior software automatically decides when to use hardware or software breakpoints
- **Software** – the CodeWarrior software always uses software breakpoints

NOTE Software breakpoints are breakpoints where the debugger writes a debug instruction into your code. These breakpoints can not be set in flash, since it is read only.

- **Hardware** – the CodeWarrior software always uses hardware breakpoints if available

NOTE Hardware breakpoints are breakpoints that use the on-chip debugging capabilities of the DSP56800E. These capabilities are limited by the number of hardware breakpoints available.

5

C for DSP56800E

This chapter contains the following sections:

- General Notes on C
- Number Formats for DSP56800E
- Calling Conventions and Stack Frames for DSP56800E
- Data Alignment Requirements for DSP56800E
- Code and Data Storage for DSP56800E
- Large Data Model Support
- Optimizing Code for DSP56800E
- Pragma Directives for DSP56800E
- Linker Issues for DSP56800E

General Notes on C

Note the following on the DSP56800E processor:

- C++ language is not supported.
- Standard C trigonometric and algebraic floating-point functions (for example, sine, cosine, tan, and square root) are not supported.

NOTE A few trigonometric or algebraic functions may be implemented within the DSP56800E MSL R2.0, but these functions are mere examples and they are not supported by the DSP56800E.

- C pointers only allow access to X memory.

Number Formats for DSP56800E

This section explains how the CodeWarrior compiler implements integer and floating-point types for 56800E processors. You can also read `limits.h` for more information on integer types and `float.h` for more information on floating-point types. Both `limits.h` and `float.h` are located under the M56800E Support folder.

56800E Integer Formats

Table 5.1 shows the sizes and ranges of the data types for the M56800E compiler.

Table 5.1 56800E Ordinal Types

Type	Option Setting	Size (bits)	Range
char	Use Unsigned Chars is disabled in the C/C++ Language settings panel	8	-128 to 127
	Use Unsigned Chars is enabled	8	0 to 255
signed char	n/a	8	-128 to 127
unsigned char	n/a	8	0 to 255
short	n/a	16	-32,768 to 32,767
unsigned short	n/a	16	0 to 65,535
int	n/a	16	-32,768 to 32,767
unsigned int	n/a	16	0 to 65,535
long	n/a	32	-2,147,483,648 to 2,147,483,647
unsigned long	n/a	32	0 to 4,294,967,295
pointer	small data model enabled	16	0 to 65,535
	large data model	24	0 to 16,777,215

56800E Floating-Point Formats

Table 5.2 shows the sizes and ranges of the floating-point types for the M56800E compiler.

Table 5.2 M56800E Floating-Point Types

Type	Size (bits)	Range
`float`	32	`1.17549e-38` to `3.40282e+38`
`short double`	32	`1.17549e-38` to `3.40282e+38`
`double`	32	`1.17549e-38` to `3.40282e+38`
`long double`	32	`1.17549e-38` to `3.40282e+38`

Calling Conventions and Stack Frames for DSP56800E

The DSP56800E compiler stores data and call functions differently than the DSP56800 compiler. Advantages of the DSP56800E include: more registers that are used for parameters and the more efficient storage of bytes.

Passing Parameters to Functions

The registers `A`, `B`, `R1`, `R2`, `R3`, `R4`, `Y0`, and `Y1` are used to pass parameters to functions. When a function is called, the parameter list is scanned from left to right.

Parameters are passed in registers in the following cases:

- The first two 16-bit integer values are passed in `Y0` and `Y1`.
- The first two 32-bit integer or float values are passed in `A` and `B`.
- The first four pointer parameters are passed in `R2, R3, R4`, and `R1` (in that order).
- If A and B are not used for 32-bit parameters, they are used for the third and fourth 16-bit parameters.
- If B is not used for a 32-bit parameter, it is used for the third 16-bit parameter.

The remaining parameters, those not passed in registers, are passed on the stack. The stack is incremented by the total amount of space required for memory parameters. The increment must be an even number of words due to the requirement that SP must be continuously long-aligned. The stack parameters are moved to the stack from left to right beginning with the stack location closest to SP. Because a long parameter must begin at an even address, one word gap may be introduced before a long parameter is moved onto the stack.

Returning Values From Functions

The registers `A`, `R0`, `R2`, and `Y0` are used to return function results as follows:

- 16-bit integer values are returned in `Y0`.
- 32-bit integer or float values are returned in `A`.
- All pointer values are returned in R2.
- Structure results are returned in a temporary space allocated by the caller. A pointer to this space is passed in R0 as a hidden parameter.
- When a function makes a dynamic allocation, R5 is reserved as a stack frame pointer. This is the original stack pointer before allocations are done.
- Registers C10 and D10 are saved across function calls. R5 is saved across calls unless it is reserved as a frame pointer (as noted above).
- Registers C2 and D2 are not saved across function calls.

Volatile and Non-Volatile Registers

Non-volatile Registers

Non-volatile registers are registers that can be saved across functions calls. These registers are also called saved over a call registers (SOCs).

Volatile Registers

Volatile registers are registers that cannot be saved across functions calls. These registers are also called non-SOC registers.

NOTE See Table 5.3 for a list of volatile (non-SOC) and non-volatile (SOC) registers.

Table 5.3 Volatile and Non-Volatile Registers

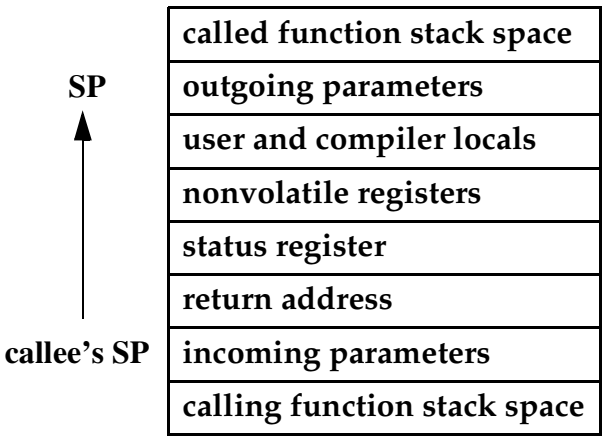
Unit	Register Name	Size	Type	Comments
Arithmetic Logic Unit (ALU)	Y1	16	Volatile (non-SOC)	
	Y0	16	Volatile (non-SOC)	
	Y	32	Volatile (non-SOC)	
	X0	16	Volatile (non-SOC)	
	A2	4	Volatile (non-SOC)	
	A1	16	Volatile (non-SOC)	
	A0	16	Volatile (non-SOC)	
	A10	32	Volatile (non-SOC)	
	A	36	Volatile (non-SOC)	
	B2	4	Volatile (non-SOC)	
	B1	16	Volatile (non-SOC)	
	B0	16	Volatile (non-SOC)	
	B10	32	Volatile (non-SOC)	
	B	36	Volatile (non-SOC)	
	C2	4	Volatile (non-SOC)	
	C1	16	Non-Volatile (SOC)	
	C0	16	Non-Volatile (SOC)	
	C10	32	Non-Volatile (SOC)	
	C	36	Volatile (non-SOC)	Includes volatile register C2.
	D2	4	Volatile (non-SOC)	
	D1	16	Non-Volatile (SOC)	
	D0	16	Non-Volatile (SOC)	
	D10	32	Non-Volatile (SOC)	
	D	36	Volatile (non-SOC)	Includes volatile register D2.

Unit	Register Name	Size	Type	Comments
Address Generation Unit (AGU)	R0	24	Volatile (non-SOC)	
	R1	24	Volatile (non-SOC)	
	R2	24	Volatile (non-SOC)	
	R3	24	Volatile (non-SOC)	
	R4	24	Volatile (non-SOC)	
	R5	24	Non-volatile (SOC)	If R5 is used as a pointer, it is no longer a non-volatile (SOC) register, that is, it can be saved over called functions.
	N	24	Volatile (non-SOC)	
	SP	24	Volatile (non-SOC)	
	N3	16	Volatile (non-SOC)	
	M01	16	Volatile (non-SOC)	In certain registers, values must be kept for proper C execution. Set to 0xFFFF for proper execution of C code.
Program Controller	PC	21	Volatile (non-SOC)	
	LA	24	Volatile (non-SOC)	
	LA2	24	Volatile (non-SOC)	
	HWS	24	Volatile (non-SOC)	
	FIRA	21	Volatile (non-SOC)	
	FISR	13	Volatile (non-SOC)	
	OMR	16	Volatile (non-SOC)	In certain registers, values must be kept for proper C execution. For example, set the CM bit.
	SR	16	Volatile (non-SOC)	
	LC	16	Volatile (non-SOC)	
	LC2	16	Volatile (non-SOC)	

Stack Frame

The stack frame is generated as shown in Figure 5.1. The stack grows upward, meaning that pushing data onto the stack increments the address in the stack pointer.

Figure 5.1 Stack Frame



The stack pointer register (SP) is a 24-bit register like other Address Generation Unit (AGU) registers and it is always treated as a word pointer. The stable position for SP during a function execution is at the top of the user and compiler locals. The SP increases during the call if the stack is used for passed parameters.

The software stack supports structured programming techniques, such as parameter passing to subroutines and local variables. These techniques can be used for both assembly language programming as well as high-level language programming. It is possible to support passed parameters and local variables for a subroutine at the same time within the stack frame.

Local data is stored by size. Smaller data is placed closest to the stack pointer in an attempt to use the various stack pointer addressing modes with small offsets. Therefore, all bytes are packed two per word near the stack pointer. The block of words is next, followed by blocks of longs. Aggregates (structs and arrays) are furthest from the stack pointer and not sorted by size.

NOTE When a function makes a dynamic allocation, R5 is reserved as a stack frame pointer. This is the stack pointer before allocations are done.

Stack Alignment

The compiler must always operate with the stack pointer long aligned. This means:

- The start-up code in the runtime first initializes the stack pointer to an odd value.
- At all times after that, the stack pointer must point to an odd word address.
- The compiler never generates an instruction that adds or subtracts an odd value from the stack pointer.
- The compiler never generates a MOVE.W or MOVEU.W instruction that uses the X:(SP)+ or X:(SP)- addressing mode.

Data Alignment Requirements for DSP56800E

The following rules describe data layout for both the stack and global memory of DSP56800E:

- Bytes are aligned on byte boundaries.

  There is an exception to this rule: bytes which are passed on the stack are always word-aligned and reside in the lower bytes.
- Words are aligned on word boundaries.
- Longs and floats (and doubles) are aligned on double-word boundaries as follows:
  - Least significant word is always on an even word address.
  - Most significant word is always on an odd word address.
  - Long accesses through pointers in AGU registers (for example, R0 through R5 or N) point to the least significant word. That is, the address is even.
  - Long accesses through pointers using SP point to the most significant word. That is, the address in SP is odd.

- Structures are aligned on a word boundary, not on a byte boundary.

NOTE A structure containing only bytes is still word aligned.

- Structures are aligned on a double-word boundary if 32-bit elements exist in the structure or if an inner structure itself is double-word aligned.
- Arrays are aligned on the size of one array element.

Pointer Types (Word versus Byte Pointers)

The use of DSP56800E byte and word pointers to implement C pointer types is implied by the alignment requirements defined above. The detailed rules used by the compiler are as follows:

- Use word pointers for all structures
- Use the SP to access the stack resident data of all types:
  - Bytes
  - Shorts
  - Longs
  - Floats
  - Doubles
  - Any pointer variables
- Use word pointers for all accesses to:
  - Shorts
  - Longs
  - Any pointer variables
- Use byte pointers for:
  - Single global or static byte variable when accessed through a pointer using X:(Rn)
  - Global or static array of byte variables

    Although no pointers are used for accessing scalar global or static byte variables directly by their addresses, an instruction with a .BP suffix is used:

```
MOVE[U].BP    X:xxxx,<dest>
MOVE.BP       <src>,X:xxxx
```

Reordering of Data for Optimal Usage

Reordering of data is subject to these guidelines:

- Reordering is required when local variables are allocated on the stack.
- Reordering is not performed for all passed parameters that are not passed in registers but instead passed in memory.
- Reordering is not performed when locating fields within a structure.

Code and Data Storage for DSP56800E

The DSP56800E has a dual Harvard architecture with separate `CODE` (P: memory) and `DATA` (X: memory) memory spaces. The ranges of memory vary depending on whether you have the small memory or large memory model enabled (Table 5.4). You might need to specify how the project-defined sections map to real memory by using the **ELF Linker and Command Language** and **M56800E Linker Settings** panel.

Table 5.4 Code and Data Memory

Section	Small Model		Large Model	
	Size	Range (Word Address)	Size	Range (Word Address)
CODE (P: memory)	128 KB	0 - 0xFFFF	1 MB	0 - 0x7FFFF
DATA (X: memory)	128 KB*	0 - 0xFFFF	32 MB	0 - 0xFFFFFF
* Character data is limited to 64 KB or 0 - 0x7FFF for small model and 16 MB or 0 - 0x7FFFFF for large model.				

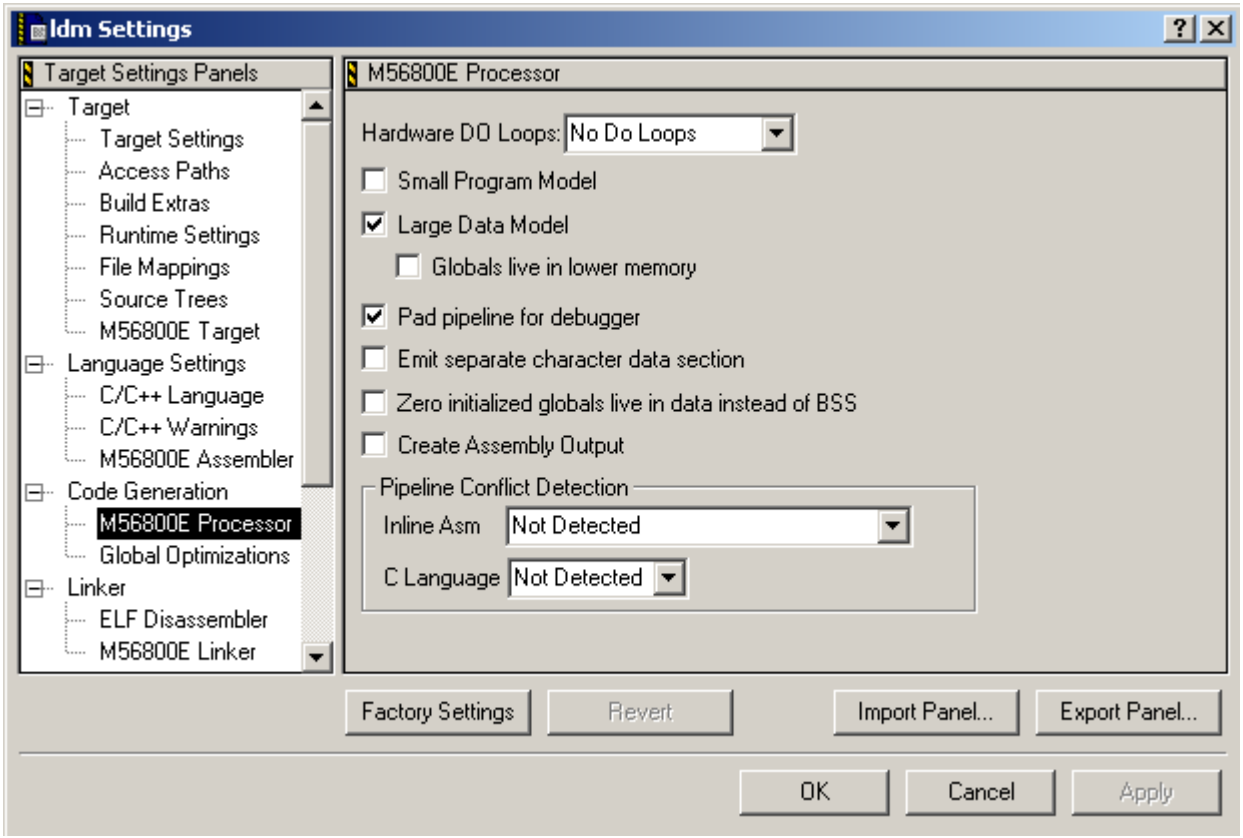
Additional Information about Character Data Addressing

An architectural peculiarity of the DSP56800E is the existence of byte addresses for character (1-byte) data and word addresses for all other data. A byte address is calculated by multiplying the word address by 2. Since an address cannot exceed the maximum physical address, if you place character data in the upper half of memory, the data is rendered unaddressable since address registers have a fixed width. For example, assuming the small memory model (maximum data address is ~64 KB words), if you place character data at 0x8001, it would require an address of 0x10002 to access it. The address 0x10002 does not fit into a 16-bit storage, as is required by the small data memory model. In order to give you more flexibility, the compiler will, under your control, place all character data into specially-named sections as described in "Emit separate character data section" on page 63.. You can then locate these sections in the lower half of the memory map, thus ensuring that the data can be addressed.

Large Data Model Support

The DSP56800E extends the data addressing range of the DSP56800 by providing 24-bit address capability to some instructions. 24-bit address modes allow the user access beyond the 64K word boundary set by 16-bit addressing. Large data memory model support is controlled by check boxes in the **M56800E Processor** panel (Figure 5.2). See "M56800E Processor" on page 61 for detail description of options in the panel.

Figure 5.2 M56800E Processor Panel: Large Data Model



Data which is located beyond the 16-bit address boundary is sometimes referred to as extended data. Extended data can be thought of as existing in extended or upper memory. Memory located below the 64K boundary is called lower memory. By default, the compiler accesses all data by 16-bit addresses. That is, absolute addresses (X:xxxx addressing mode) are limited to 16 bits. Direct addressing or pointer registers load or store 16-bit addresses. Indexed addressing indexes are 16-bit quantities. Data pointers are treated as 16-bit pointers and you may store them in a single word of memory.

When the large data memory model is enabled, the compiler accesses all data by 24-bit addressing modes. Data pointers are treated as 24-bit quantities and you may store them in 2 words of memory. Absolute addressing occurs as 24-bit absolute addresses. Thus, you may access the entire 24-bit data memory and choose to locate data objects anywhere in memory.

You are not required to change C source code to take advantage of the large data memory model.

Examples in DSP56800E assembly code of extended data addressing are:

Examples: Extended Data Addressing

```
move.w x:0x123456,A1     ; move int using 24 bit absolute address
tst.l   x:(R0-0x123456) ; test a global long for zero using 24-bit pointer
                         ; indexed addressing
move.l r0,x:(R0)+        ; r0 stored as 32-bit quantity
cmpa r0,r1               ; compare pointer registers as 24 bit quantities
```

While the large data memory model is convenient because you can place data objects anywhere in the 24-bit data memory map, it is inefficient because extended data addressing requires more program memory and execution cycles. Many target applications' entire global and static data will reside comfortably within the 64 KB word memory boundary. With this in mind, you can use the **Globals live in lower memory** option (see Figure 5.2 on page 91) to provide the flexibility of the large data memory model with the efficiency of the small data model access to globals and statics.

The **Globals live in lower memory** preference panel selection instructs the compiler to access global and static data with 16-bit addresses, while all pointer and stack operations are performed using 24-bit addressing. Global data may still exist in extended memory if you place the global data in extended memory using the linker control file. But, you can only access global data through data pointers or assembly level addressing.

NOTE It is your responsibility to properly place data in the memory map if the Globals live in lower memory option is enabled. If Globals live in lower memory is enabled, the run time results are undefined when global data exists and accessed in extended memory.

Examples: Accessing Data Objects

Examples of how the compiler might generate accesses to various data objects are:

Example 1

This example (Table 5.5) demonstrates how a global integer is accessed with the various combinations of the large data memory model switches. The global variable is located at address X:0x1234.

```
Int g1;
```

Table 5.5 Settings for Accessing Global Integer

Large Data Memory Model	Globals in 64K memory	Instruction	Comments
Off	Off	move.w X:0x1234,y0	Default values
On	Off	move.w X:0x001234,y0	
Off	On	Combination not allowed	
On	On	move.w X:0x1234,y0	Globals accessed using 16-bit addressing

Example 2

This example (Table 5.6 on page 94) demonstrates how a global pointer variable is loaded into an address register. The pointer variable is located at X:0x4567.

```
Int * gp1;
```

Table 5.6 Settings for Loading a Global Pointer Variable

Large Data Memory Model	Globals in 64K memory	Instruction	Comments
Off	Off	move.w X:0x4567,r0	Default 16-bit addressing, 16-bit pointer value
On	Off	move.l X:0x004567,r0	24-bit addressing, pointer value is 24-bit
Off	On	Combination not allowed	
On	On	move.l X:0x4567,r0	16-bit addressing, pointer value is 24-bit

External Library Compatibility

If the main application is built with the large data model, external libraries written in C must be built with the large data model enabled. This requirement is enforced by the linker. The linker will catch issues where a global object is located out of range of a particular instruction encoding. A more serious compatibility problem exists regarding passing pointer parameters. Applications built with the large data memory model may pass pointer parameters on the stack. Libraries built using the small memory model may expect the pointer arguments to occupy a single word of memory, when in fact the arguments occupy 2 words. This will lead to a runtime corruption of the stack.

You may or may not build external libraries or modules written in assembly with extended addressing modes. The linker does not enforce any compatibility rules on assembly language modules or libraries.

To prevent compatibility issues between C object files built with the large data memory model and those built with the small data memory model, the memory model is encoded into the object file. The linker verifies that all objects linked into an executable have compatible memory models. The ELF header's e_flags field contains bit fields in which encode the data memory model attributes of the object file.

```
#define EF_M56800E_LDMM 0x00000001 /* Large data memory model flag */
```

Additionally, C language objects are identified by an ELF header flag.

```
#define EF_M56800E_C 0x00000002 /* Object file generated from C source */
```

Optimizing Code for DSP56800E

This section explains register coloring, an optimization specific to DSP56800E development.

The C compiler performs register coloring. In this optimization, the compiler may assign two or more register variables to the same register. The compiler does this optimization if the code is not using the variables at the same time. In the following example, the compiler can place `i` and `j` in the same register:

Listing 5.1 Sample Code - Register Coloring

```
short i;
int j;

for (i=0; i<100; i++) { MyFunc(i); }
for (j=0; j<100; j++) { MyFunc(j); }
```

However, if the code includes the expression `MyFunc (i+j)`, it indicates to the compiler that `i` and `j` are simultaneously live, and places the variables in different registers.

With the DSP56800E development tools, you can instruct the compiler to:

1. **Store all local variables on the stack.**

   The compiler loads and stores local variables when you read and write to them. Level 0 behavior may be preferable during debugging because all variables are guaranteed to have meaningful values after they have been initialized through the end of the function.

2. **Place as many local variables as possible in registers.**

   In this case, two or more variables whose lifetimes do not overlap can use the same register. If you prefer this behavior, set the optimization to Level 1 or higher in the **Global Optimizations** panel.

NOTE At optimization Level 1 and higher, local variables can be assigned to different registers in different sections of your code. This behavior could produce unexpected results if you compile your code and then attempt to debug it.

Variables that are declared `volatile` (or those that have the address taken) are not kept in registers and may be useful in the presence of interrupts.

3. **Run Peephole Optimization for the DSP56800E at Levels 1 through 4.**

NOTE Peephole optimizations are small, local optimizations that eliminate some compare instructions and improve branch sequences.

Pragma Directives for DSP56800E

This section describes the pragma options available with the Metrowerks C compiler.

A pragma is a method for modifying compiler settings from the source code rather than the preference panels. Typically, you would use the settings panels to set global options and use pragmas for special cases. The setting can be easily turned on and off for different portions of the code.

asmoutput

Description — This pragma controls the generation of an assembly file for each compiled file processed.

Prototype — `#pragma asmoutput [on|off]`

Remarks — Each file that has assembly output activated will generate an `.asm` file that is equivalent to the original `.c` file. The `.asm` file is created in the same directory as the original `.c` file, and replaces the `.c` extension with `.asm`.

This pragma works for an individual file. However, an option is also available in the **M56800E Processor** panel to globally create assembly output.

check_c_src_pipeline

Description This pragma controls the detection of a pipeline conflict in the C language source code when it is compiled.

Prototype `#pragma check_c_src_pipeline [off|conflict]`

Remarks Use this pragma to detect and generate an error message if the C language source code has a pipeline conflict when it is compiled. In some cases, the source code can be a mix of assembly language and C language.

The option `conflict` only detects and generates error messages for pipeline conflict.

check_inline_asm_pipeline

Description This pragma controls the detection of a pipeline conflict in the assembly language source code when it is compiled.

Prototype `#pragma check_inline_asm_pipeline [off|conflict|conflict_and_stall]`

Remarks Use this pragma to detect and generate an error message if the assembly language source code has a pipeline conflict when it is compiled. In some cases, the source code can be a mix of assembly language and C language.

The option `conflict` only detects and generates error messages for pipeline conflict.

The option `conflict_and_stall` detects and generates error messages for pipeline conflicts and stalls.

interrupt

Description This pragma controls the compilation of object code for interrupt routines.

Prototype `#pragma interrupt [<options>] [<mode>] [on|off|reset]`

Remarks An Interrupt Service Routine (ISR) is a routine that is executed when an interrupt occurs. Setting C routines as ISRs is done using

pragmas (`pragma interrupt`). To make a routine service an interrupt, you must:

- Write the routine.
- Set up the routine so that it is called when some interrupt occurs.

The `pragma interrupt` option can be used to perform the following:

- Instruct the compiler to push register values on the software stack at entry to a C function and restore them upon exit.
- Preserve the register values for the function that was interrupted.
- Emit an RTI for the return statement depending upon the mode selected. If the interrupt routine has a return value, the return register is not saved.

There are several ways to use this pragma, with an on | off | reset arguments, or with no arguments.

Arguments

<options>

alignsp	Aligns the stack pointer register correctly to allow long values to be pushed on to the stack. Use this option when your project mixes C code and assembly code. Use this option specifically on ISRs which may interrupt assembly routines that do not maintain the long stack alignment requirements at all times. Restores the stack pointer to its original value before returning from the subroutine.
comr	The Operating Mode Register (OMR) is set for the following to ensure correct execution of C code in the ISR: 36-bit values used for condition codes. (CM bit cleared) Convergent Rounding. (R bit cleared) No Saturation mode. (SA bit cleared) Instructions fetched from P memory. (XP bit cleared)

<mode>

saveall	Preserves register values by saving and restoring all registers by calling the INTERRUPT_SAVEALL and INTERRUPT _ RESTOREALL routines in the Runtime Library.

called — Preserves register values by saving and restoring registers used by the routine. The routine returns with an RTS. Routines with pragma interrupt enabled in this mode are safe to be called by ISRs.

default — This is the mode when no mode is specified. In this mode, the routine preserves register values by saving and restoring the registers that are used by the routine. The routine returns with an RTI.

on | off | reset

on — Enables the option to compile all C routines as interrupt routines.

off — Disables the option to compile all C routines as interrupt routines.

reset — Restores the option to its previous setting.

NOTE Use on or off to change the pragma setting, and then use reset to restore the previous pragma setting.

Listing 5.2 Sample Code - #pragma interrupt on | off | reset

```
#pragma interrupt off// To be used as default value
// Non ISR code
#pragma interrupt on
void ISR_1(void) {
  // ISR_1 code goes here.
}

void ISR_2(void) {
  // ISR_2 code goes here.
}
#pragma interrupt reset
```

If the pragma is inside a function block, compile the current routine as an interrupt routine. If the pragma is not inside a function block, compile the next routine as an interrupt routine. To disable the pragma, use `#pragma interrupt off` after `#pragma interrupt` (Listing 5.3).

Listing 5.3 Sample Code - #pragma interrupt and function block

```
// Non ISR code
void ISR_1(void) {
#pragma interrupt
  // ISR_1 code goes here.
}
#pragma interrupt
void ISR_2(void) {
  // ISR_2 code goes here.
}
```

See Listing 5.4 for an example of using the 'called' option in the interrupt pragma.

Listing 5.4 Sample Code - using the ‘called’ option in # pragma interrupt

```
extern long Data1, Data2, Datain;

void ISR1_inc_Data1_by_Data2(void)
{
/* This is a routine called by the interrupt service routine
ISR1(). */
#pragma interrupt called
Data1+=Data2;
return;
}

void ISR1(void)
{
/* This is an interrupt service routine. */
#pragma interrupt
Data2 = Datain+2;
ISR_inc_Data1_by_Data2();
}
```

Avoiding Possible Hitches with enabled Pragma Interrupt

Enabling pragma interrupt with the *called* or *default* mode for a C routine saves only the volatile registers for that C routine. Register values are not preserved if the ISR makes one or more function calls. You might want to avoid the situations described below:

- If a routine that has pragma interrupt enabled (caller) calls another C function/routine (callee), it is possible that the callee can change some registers that are not saved by the caller. To avoid this, use either of the following options:
  - Call only pragma interrupt enabled routines from routines that are pragma interrupt enabled using the *called* mode, or
  - Use the pragma interrupt *saveall* mode for the caller.

  The first option may be more efficient because only the registers that are used are preserved. The second option is easier to implement, but is likely to have a large overhead.
- The situation described above also holds true for library functions because library functions do not have pragma interrupt enabled. These calls include: C Standard Library calls and Runtime Library calls (such as multiplication, division and floating point math).
- C Standard Library and Runtime Library (CW libraries) functions require the AGU (Address Generation Unit) to be in linear addressing mode, that is, the M01 registers are set to -1. If a function is interrupted and was using modulo address arithmetic, any calls to CW libraries from the ISR do not work unless the M01 is set to -1 in the ISR. Also, the M01 register would need to be restored before exiting the ISR so that the interrupted function can resume as before, with the same modulo address arithmetic mode settings.

optimization_level

Description This pragma controls global optimization.

Prototype `#pragma optimization_level [<level> | reset]`

Remarks `<level> is an integer number between 0 and 4 inclusive`

The optimization_level pragma controls the global optimization level programmatically through the #pragma preprocessor statement. You may set the optimization level to any legal value (0-4) using this method. The 'reset' option resets the optimization level to its prior value before the #pragma is encountered. This pragma does not affect the interactive preference panel settings for global optimization levels. If this pragma occurs in the middle of a function definition, the entire function is compiled as if the pragma had occurred before the function definition since the actual compilation of the function is deferred until the function is parsed entirely.

Listing 5.5 Sample Code - Pragma Optimization

```
// Function definition
int afunc ( void )
{
// Function statements...
#pragma optimization_level 0
// Remaining function statements...
}
// Restore optimization level to its previous value
#pragma optimization_level reset

// The entire function 'afunc' is compiled under optimization
level 0.
```

optimize_for_size

Description — This pragma controls optimization to reduce the size of object code.

Prototype — `#pragma optimize_for_size on|off|reset`

Remarks — This option lets you choose what the compiler does when it must decide between creating small code or fast code. If this option is on, the compiler creates smaller object code at the expense of speed. If this option off, the compiler creates faster object code at the expense of size.

This pragma corresponds to the **Optimize for Size** option in the **Global Optimizations** panel.

Listing 5.6 Sample Code - pragma optimize_for_size

```
// Function definition
int afunc ( void )
{
// Function statements...
#pragma optimize_for_size on
// Remaining function statements...
}
// Restore optimize_for_size to its previous value
#pragma optimize_for_size reset

// The entire function 'afunc' is compiled and optimized for size.
```

define_section

Description This pragma controls the definition of a custom section.

Prototype
```
#pragma define_section <sectname> <istring> [
<ustring> ] [ <accmode> ]
```

Remarks Arguments:

`<sectname>`

Identifier by which this user-defined section is referenced in the source, that is, via the following instructions:

- `#pragma section <sectname> begin`
- `__declspec(<sectname>)`

`<istring>`

Section name string for initialized data assigned to <section>.

For example:

`".data"`

`Optional Arguments:`

`<ustring>`

Section name string for uninitialized data assigned to <section>. If `ustring` is not specified than `istring` is used.

`<accmode>`

One of the following indicating the attributes of the section:

`R`	readable
`RW`	readable and writable (Note, this is the default)
`RX`	readable and executable
`RWX`	readable, writable, and executable

Note The default is `RW`.

NOTE For an example of `define_section`, see Listing 5.7.

Related Pragma `section`

section

Description This pragma controls the organization of object code.

Prototype
```
#pragma section <sectname> begin
[...data..]
#pragma section <sectname> end
```

Remarks Argument:

`<sectname>`

Identifier by which this user-defined section is referenced in the source.

Listing 5.7 Sample Code -pragma define_section and pragma section

```
/* 1. Define the section */
#pragma define_section mysection ".mysection.data" RW

/* 2. Specify the data to be put into the section. */
#pragma section mysection begin
int a[10] = {'0','1','2','3','4','5','6','7','8','9'};
int b[10];
#pragma section mysection end

int main(void) {
    int i;
    for (i=0;i<10;i++)
      b[i]=a[i];
}

/* 3. In the linker command file, add the section to the
".main_application_data"  entries in the ".data sections" area of
the linker command file by inserting the following line:
    * (.mysection.data)

*/
```

Related Pragma `define_section`

unused

Description — This pragma controls the suppression of warnings for variables and parameters that are not referenced in a function.

Prototype — `#pragma unused ( var_name [, var_name]... )`

Remarks — This pragma suppresses the compile time warnings for the unused variables and parameters specified in its argument list. You can use this pragma only within a function body, and the listed variables must be within the scope of the function.

Listing 5.8 Sample Code -pragma unused

```
#pragma warn_unusedvar on  /* This pragma is equivalent to "Unused
                              Variables" option in C/C++ Warning
                              panel */
#pragma warn_unusedarg on  /* This pragma is equivalent to "Unused
                              Variables" option in C/C++ Warning
                              panel */
static void ff(int a)
{

int b;
#pragma unused(a,b)        /* Compiler does not warn that a and b
                              are unused */

}
```

Linker Issues for DSP56800E

This section explains background information on the DSP56800E linker and also how it operates.

Deadstripping Unused Code and Data

The M56800E Linker deadstrips unused code and data only from files compiled by the CodeWarrior C compiler. Assembler relocatable files and C object files built by other compilers are never deadstripped.

Libraries built with the CodeWarrior C compiler contribute only the used objects to the linked program. If a library has assembly or other C compiler-built files, only those files that have at least one referenced object contribute to the linked program. If you enable deadstripping, the linker always completely ignores unreferenced objects files.

Link Order

The DSP56800E linker always processes C and assembly source files, as well as archive files (.a and .lib), in the order specified under the **Link Order** tab in the project window. Therefore, if a symbol is defined in a source-code file and a library, the linker uses the definition which appears first in the link order.

If you want to change the link order, select the **Link Order** tab in the project window and drag your source or library file to the preferred location in the link order list. Files that appear at the top of the list are linked first.

6

Inline Assembly Language and Intrinsics

This chapter explains the support for inline assembly language and intrinsic functions that is built into the CodeWarrior™ compiler. This chapter only covers the CodeWarrior IDE implementation of Motorola assembly language.

Working With DSP56800E Inline Assembly Language

This section explains how to use the CodeWarrior compiler's for inline assembly language programming, including assembly language syntax.

This chapter contains the following sections:

- Working With DSP56800E Inline Assembly Language
- Inline Assembly Optimization
- Calling Functions from Assembly Language
- Instrinsic Functions for DSP56800E

Inline Assembly Language Syntax for DSP56800E

This section explains the inline assembly language syntax specific to DSP56800E development with the CodeWarrior IDE.

Function-level Inline Assembly Language

NOTE Although the DSP56800 and DSP56800E are assembly source code compatible, do not re-use DSP56800 assembly code in the DSP56800E compiler due to calling convention differences between the DSP56800 and DSP56800E compilers.

To specify that a block of code in your file should be interpreted as assembly language, use the `asm` keyword and standard DSP56800E instruction mnemonics.

To ensure that the C compiler recognizes the `asm` keyword, you must disable the **ANSI Keywords Only** option in the **C/C++ Language** panel.

You can use the M56800E inline assembly language to specify that an *entire function* is in assembly language by using the syntax displayed in Listing 6.1.

Listing 6.1 Function-level Syntax

```
asm <function header>
{
   <local declarations>
       <assembly instructions>
}
```

The function header is any valid C function header, and the local declarations are any valid C local declarations.

Statement-level Inline Assembly Language

The M56800E inline assembly language supports single assembly instructions as well as `asm` blocks, *within* a function using the syntax in Listing 6.2.

Listing 6.2 Statement-level Syntax

```
asm { inline assembly statement
      inline assembly statement
        ...
}

asm (inline assembly statement) ;
```

The inline assembly language statement is any valid assembly language statement. Note that the only difference between the two lines of code is the use of braces versus parenthesis and the terminating semicolon requirement when using parenthesis.

Adding Assembly Language to C Source Code

There are two ways to add assembly language statements in a C source code file. You can define a function with the `asm` qualifier, or you can use the inline assembly language.

The first method uses the `asm` keyword to specify that *all* statements in the function are in assembly language, as shown in Listing 6.3 and Listing 6.7. Note that if you are using this method, you must define local variables within the function.

Listing 6.3 Defining a Function with `asm`

```
asm long MyAsmFunction(void)
{
   /* Local variable definitions */
   /* Assembly language instructions */
}
```

The second method uses the `asm` qualifier as a statement to provide inline assembly language instructions, as shown in Listing 6.4. Note that if you are using this method, you must *not* define local variables within the inline `asm` statement.

Listing 6.4 Inline Assembly with `asm`

```
long MyInlineAsmFunction(void)
{
    asm { move  x:(r0)+,x0 }
}
```

Assembly Language Quick Guide

Keep these points in mind as you write assembly language functions:

- All statements must either be a label:

  `[`*`LocalLabel`*`:]`

  Or an instruction:

  `( (`*`instruction`*`) [`*`operands`*`] )`

- Each statement must end with a new line
- Assembly language directives, instructions, and registers are not case-sensitive. The following two statements are the same:

```
add   x0,y0
ADD   X0,Y0
```

Creating Labels for M56800E Assembly

A label can be any identifier that you have not already declared as a local variable. A label must end with a colon.

Listing 6.5 Labels in M56800E Assembly

```
x1:  add  x0,y1,a
x2:
     add  x0,y1,a
x3   add  x0,y1,a  //ERROR, MISSING COLON
```

Using Comments in M56800E Assembly

Comments in inline assembly language can only be in the form of C and C++ comments. You cannot begin the inline assembly language comments with a semicolon (;) nor with a pound sign (#) - the preprocessor uses the pound sign. You can use the semicolon for comments in `.asm` sources. The proper comment format is shown in Listing 6.6.

Listing 6.6 Comments Allowed in M56800E In-line Assembly Language

```
move    x:(r3),y0       #  ERROR
add     x0,y0           // OK
move    r2,x:(sp)       ; ERROR
adda    r0,r1,n         /* OK */
```

Inline Assembly Optimization

To optimize a specific block of inline assembly source code, use the inline assembly directive `.optimize_iasm [on|off]`. By default this directive is off, but can be turned on for sections of code. If you do not specifically specify this directive as off (for example, `.optimize_iasm off)` then optimizations will continue until the end of the function.

Calling Assembly Language Functions from C Code

You can call assembly language functions from C just like you would call any standard C function. You need to use standard C syntax for calling inline assembly language functions and pure assembly language functions in `.asm` files.

Calling Inline Assembly Language Functions

You can call inline assembly language functions just like you would call any standard C function. Listing 6.7 demonstrates how to create an inline assembly language function in a C source file. This example adds two 16-bit integers and returns the result.

Notice that you are passing two 16-bit addresses to the `add_int` function. You pick up those addresses in R2 and R3, and in Y0 pass back the result of the addition.

Listing 6.7 Sample Code - Creating an Inline Assembly Language Function

```
asm int add_int( int * i, int * j )
{
  move.w    x:(r2),y0
  move.w    x:(r3),x0
  add   x0,y0
  // int result returned in y0
  rts
}
```

Now you can call your inline assembly language function with standard C notation, as in Listing 6.8.

Listing 6.8 Sample Code - Calling an Inline Assembly Language Function

```
  int x = 4, y = 2;

  y = add_int( &x, &y ); /* Returns 6 */
```

Calling Pure Assembly Language Functions

In order for your assembly language files to be called from C code, you need to specify a `SECTION` mapping for your code so that it is linked appropriately. You must also specify a memory space location. Code is usually specified to program memory (P) space with the `ORG` directive.

When defining an assembly language function, use the `GLOBAL` directive to specify the list of symbols within the current section that needs to be accessible by any of the other sections. You can then define the assembly language function.

An example of a complete assembly language function is shown in Listing 6.9. In this function, two 16-bit integers are written to program memory. A separate function is needed to write to P:

memory because C pointer variables cannot be employed. C pointer values only allow access to X: data memory.

The first parameter is a short value and the second parameter is the 16-bit address where the first parameter is written.

Listing 6.9 Sample Code - Creating an Assembly Language Function

```
                                ;"my_asm.asm"
  SECTION user                  ;map to user defined section in CODE
  ORG P:                        ;put the following program in P
                                ;memory

  GLOBAL  Fpmemwrite            ;This symbol is defined within the
                                ;current section and should be
                                ;accessible by all sections
Fpmemwrite:
  MOVE    Y1,R0                 ;Set up pointer to address
  NOP                           ;Pipeline delay for R0
  MOVE    Y0,P:(R0)+            ;Write 16-bit value to address
                                ;pointed to by R0 in P: memory and
                                ;post-increment R0
  rts                           ;return to calling function

  ENDSEC                        ;End of section
  END                           ;End of source program
```

You can now call your assembly language function from C, as shown in Listing 6.10.

Listing 6.10 Sample Code - Calling an Assembly Language Function from C

```
void pmemwrite( short, short );/* Write a value into P: memory */

void main( void )
{
  // ...other code

  // Write the value given in the first parameter to the address
  // of the second parameter in P: memory
  pmemwrite( (short)0xE9C8, (short)0x0010 );

  // other code...
}
```

Calling Functions from Assembly Language

Assembly programs can call C function or Assembly language functions. This section explains the compiler convention for:

- Calling C Functions from Assembly Language

  Functions written in C can be called from within assembly language instructions. For example, if you defined your C program function as:

  ```
  void foot( void ) {
       /* Do something */
  }
  ```

  You could then call your C function from assembly language as:

  ```
  jsr  Ffoot
  ```

- Calling Assembly Language Functions from Assembly Language

  To call an assembly language function from assembly language, use the `jsr` instruction with the function name as defined in your assembly language source. For example, you can call your function in Listing 6.9 on page 117 as:

  ```
  jsr  Fpmemwrite
  ```

Instrinsic Functions for DSP56800E

This section explains issues related to DSP56800E intrinsic functions and using them with DSP56800E projects.

An Overview of Intrinsic Functions

CodeWarrior for DSP56800E has intrinsic functions to generate inline assembly language instructions.

Intrinsic functions are used to target specific processor instructions. They can be helpful in accomplishing a few different things:

- Intrinsic functions let you pass in data to perform specific optimized computations. For example, some calculations may be inefficient if coded in C because the compiler has to follow ANSI C rules to represent data, and this may cause the program to jump to runtime math routines for certain computations. In such cases, it probably is better to code these calculations using assembly language instructions and intrinsic functions.
- Intrinsic functions can control small tasks. For example, with intrinsic functions you can set a bit in the operating mode register to enable saturation. This is more convenient than using inline assembly language syntax and specifying the operation in an `asm` block, every time that the operation is required.

NOTE Support for intrinsic functions is not part of the ANSI C standard. They are an extension provided by the CodeWarrior compiler.

Implementation

The CodeWarrior DSP56800E intrinsic functions are implemented as inline C functions in `intrinsics_56800E.h`, which is located in the MSL directory tree. These inline functions contain mostly inline assembly language code. An example is the abs_s intrinsic. It is defined as:

Listing 6.11 Example Code - Definition of Intrinsic Function: abs_s

```
#define    abs_s(a) __abs_s(a)
           /* ABS_S */
inline Word16 __abs_s(register Word16 svar1)
{
/*
 *   Defn: Absolute value of a 16-bit integer or fractional value
 *         returning a 16-bit result.
 *         Returns $7fff for an input of $8000
 *
 *   DSP56800E instruction syntax:  abs FFF
 *              Allowed src regs:  FFF
 *              Allowed dst regs:  (same)
 *
 *   Assumptions: OMR's SA bit was set to 1 at least 3 cycles
 *   before this code.
 */
   asm(abs svar1);
   return svar1;
}
```

Fractional Arithmetic

Many of the intrinsic functions for Motorola DSP56800E use fractional arithmetic with implied fractional values. An implied fractional value is a symbol, which has been declared as an integer type, but is to be calculated as a fractional type. Data in a memory location or register can be interpreted as fractional or integer, depending on the needs of a user's program.

All intrinsic functions that generate multiply and divide instructions (DIV, MPY, MAC, MPYR, and MACR) perform fractional arithmetic on implied fractional values. The following equation shows the relationship between a 16-bit integer and a fractional value:

$$\text{Fractional Value} = \text{Integer Value} / (2^{15})$$

Similarly, the equation for converting a 32-bit integer to a fractional value is as follows:

$$\text{Fractional Value} = \text{Long Integer Value} / (2^{31})$$

 shows how both 16 and 32-bit values can be interpreted as either fractional or integer values.

Table 6.1 Interpretation of 16- and 32-bit Values

Type	Hex	Integer Value	Fixed-point Value
short int	0x2000	8192	0.25
short int	0xE000	-8192	-0.25
long int	0x20000000	536870912	0.25
long int	0xE0000000	-536870912	-0.25

Macros Used with Intrinsics

These macros are used in intrinsic functions:

- Word16. A macro for signed short.
- Word32. A macro for signed long.

Intrinsic Functions for Math Support

The intrinsic functions for math support supported by the DSP56800E are shown in Figure 6.1 and the intrinsic functions for modulo addressing are given in "Modulo Addressing Intrinsic Functions: Definitions and Examples" on page 158. However, please refer to `intrinsics_56800E.h` for any last minute changes.

Figure 6.1 Intrinsic Functions for DSP56800E

Category	Function	Category (cont.)	Function (cont.)
Absolute/Negate	abs_s	Multiplication/ MAC	mac_r
	negate		msu_r
	L_abs		mult
	L_negate		mult_r
Addition/Subtraction	add		L_mac
	sub		L_msu
	L_add		L_mult
	L_sub		L_mult_ls
Control	stop	Normalization	ffs_s
	wait		norm_s
	turn_off_conv_rndg		ffs_l
	turn_off_sat		norm_l
	turn_on_conv_rndg	Rounding	round
	turn_on_sat	Shifting	shl
Deposit/ Extract	extract_h		shlftNs
	extract_l		shlfts
	L_deposit_h		shr
	L_deposit_l		shr_r
Division	div_s		shrtNs
	div_s4q		L_shl
	div_ls		L_shlftNs
	div_ls4q		L_shlfts
			L_shr
			L_shr_r
			L_shrtNs

Absolute/Negate

Click any of the following links to jump to the corresponding intrinsic functions:

- abs_s
- negate
- L_abs
- L_negate

abs_s

Definition — Absolute value of a 16-bit integer or fractional value returning a 16-bit result. Returns 0x7FFF for an input of 0x8000.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 abs_s(Word16 svar1)
```

Example

```
int result, s1 = 0xE000;  /*  - 0.25    */
result = abs_s(s1);
// Expected value of result: 0x2000 = 0.25
```

negate

Definition — Negates a 16-bit integer or fractional value returning a 16-bit result. Returns 0x7FFF for an input of 0x8000.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 negate(Word16 svar1)
```

Example

```
int result, s1 = 0xE000;  /*  - 0.25    */
result = negate(s1);
// Expected value of result: 0x2000 = 0.25
```

L_abs

Definition — Absolute value of a 32-bit integer or fractional value returning a 32-bit result. Returns 0x7FFFFFFF for an input of 0x80000000.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_abs(Word32 lvar1)
```

Example

```
long result, l = 0xE0000000;  /*  - 0.25    */
result = L_abs(s1);
// Expected value of result: 0x20000000 = 0.25
```

L_negate

Definition — Negates a 32-bit integer or fractional value returning a 32-bit result. Returns 0x7FFFFFFF for an input of 0x80000000.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_negate(Word32 lvar1)
```

Example

```
long result, l = 0xE0000000;  /*  - 0.25    */
result = L_negate(s1);
// Expected value of result: 0x20000000 = 0.25
```

Addition/Subtraction

Click any of the following links to jump to the corresponding intrinsic functions:

- add
- sub
- L_add
- L_sub

---

add

Definition — Addition of two 16-bit integer or fractional values, returning a 16-bit result.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 add(Word16 src_dst, Word16 src2)
```

Example

```
short s1 = 0x4000;  /* 0.5  */
short s2 = 0x2000;  /* 0.25 */
short result;

result = add(s1,s2);
// Expected value of result: 0x6000 = 0.75
```

sub

Definition — Subtraction of two 16-bit integer or fractional values, returning a 16-bit result.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 sub(Word16 src_dst, Word16 src2)
```

Example

```
short s1 = 0x4000;  /* 0.5   */
short s2 = 0xE000;  /* -0.25 */
short result;

result = sub(s1,s2);
// Expected value of result: 0x6000 = 0.75
```

L_add

Definition — Addition of two 32-bit integer or fractional values, returning a 32-bit result.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_add(Word32 src_dst, Word32 src2)
```

Example

```
long la = 0x40000000;  /* 0.5  */
long lb = 0x20000000;  /* 0.25 */
long result;

result = L_add(la,lb);
// Expected value of result: 0x60000000 = 0.75
```

L_sub

Definition — Subtraction of two 32-bit integer or fractional values, returning a 32-bit result.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_sub(Word32 src_dst, Word32 src2)
```

Example

```
long la = 0x40000000;  /* 0.5   */
long lb = 0xE0000000;  /* -0.25 */
long result;

result = L_sub(la,lb);
// Expected value of result: 0x60000000 = 0.75
```

Control

Click any of these links to jump to the corresponding intrinsic functions:

- stop
- wait
- turn_off_conv_rndg
- turn_off_sat
- turn_on_conv_rndg
- turn_on_sat

---

stop

Definition: Generates a STOP instruction which places the processor in the low power STOP mode.

Prototype:
```
void  stop(void)
```

Usage:
```
stop();
```

---

wait

Definition: Generates a WAIT instruction which places the processor in the low power WAIT mode.

Prototype:
```
void  wait(void)
```

Usage:
```
wait();
```

turn_off_conv_rndg

Definition: Generates a sequence for disabling convergent rounding by setting the R bit in the OMR register and waiting for the enabling to take effect.

Note: When convergent rounding is disabled, 2's complement rounding is then used when rounding is performed.

Prototype:
```
void  turn_off_conv_rndg(void)
```

Usage:
```
turn_off_conv_rndg();
```

turn_off_sat

Definition: Generates a sequence for disabling automatic saturation in the MAC Output Limiter by clearing the SA bit in the OMR register and waiting for the disabling to take effect.

Prototype:
```
void  turn_off_sat(void)
```

Usage:
```
turn_off_sat();
```

turn_on_conv_rndg

Definition: Generates a sequence for enabling convergent rounding by clearing the R bit in the OMR register and waiting for the enabling to take effect.

Prototype:
```
void  turn_on_conv_rndg(void)
```

Usage:
```
turn_on_conv_rndg();
```

turn_on_sat

Definition — Generates a sequence for enabling automatic saturation in the MAC Output Limiter by setting the SA bit in the OMR register and waiting for the enabling to take effect.

Prototype
```
void  turn_on_sat(void)
```

Usage
```
turn_on_sat();
```

Deposit/ Extract

Click any of the following links to jump to the corresponding intrinsic functions:

- extract_h
- extract_l
- L_deposit_h
- L_deposit_l

---

extract_h

Definition — Extracts the 16 MSBs of a 32-bit integer or fractional value. Returns a 16-bit value. Does not perform saturation. When an accumulator is the destination, zeroes out the LSP portion. Corresponds to "truncation" when applied to fractional values.

Prototype

```
Word16 extract_h(Word32 lsrc)
```

Example

```
long l = 0x87654321;
short result;

result = extract_h(l);
// Expected value of result: 0x8765
```

---

extract_l

Definition — Extracts the 16 LSBs of a 32-bit integer or fractional value. Returns a 16-bit value. Does not perform saturation. When an accumulator is the destination, zeroes out the LSP portion.

Prototype

```
Word16 extract_l(Word32 lsrc)
```

Example

```
long l = 0x87654321;
short result;

result = extract_l(l);
// Expected value of result: 0x4321
```

L_deposit_h

Definition — Deposits the 16-bit integer or fractional value into the upper 16 bits of a 32-bit value, and zeroes out the lower 16 bits of a 32-bit value.

Prototype

```
Word32 L_deposit_h(Word16 ssrc)
```

Example

```
short s1 = 0x3FFF;
long result;

result = L_deposit_h(s1);
// Expected value of result: 0x3fff0000
```

L_deposit_l

Definition — Deposits the 16-bit integer or fractional value into the lower 16 bits of a 32- bit value, and sign extends the upper 16 bits of a 32-bit value.

Prototype

```
Word32 L_deposit_l(Word16 ssrc)
```

Example

```
short s1 = 0x7FFF;
long result;

result = L_deposit_l(s1);
// Expected value of result: 0x00007FFF
```

Division

Click any of the following links to jump to the corresponding intrinsic functions:

- div_s
- div_s4q
- div_ls
- div_ls4q

---

div_s

Definition — Single quadrant division, that is, both operands positive of two 16-bit fractional values, returning a 16-bit result. If both operands are equal, returns 0x7FFF (occurs naturally).

Note — Does not check for division overflow cases.

Does not check for divide by zero cases.

Prototype

```
Word16 div_s(Word16 s_denominator, Word16
s_numerator)
```

Example

```
short s1=0x2000; /*  0.25 */
short s2=0x4000; /*  0.5  */
short result;

result = div_s(s2,s1);
// Expected value of result: 0.25/0.5 = 0.5 =
0x4000
```

div_s4q

Definition — Four quadrant division of two 16-bit fractional values, returning a 16-bit result.

Note — Does not check for division overflow cases.

Does not check for divide by zero cases.

Prototype
```
Word16 div_s4q(Word16 s_numerator, Word16
s_denominator)
```

Example
```
short s1=0xE000;/*  -0.25 */
short s2=0xC000;/*  -0.5  */
short result;

result = div_s4q(s1,s2);
// Expected value of result: -0.25/-0.5 = 0.5 =
0x4000
```

div_ls

Definition — Single quadrant division, that is, both operands positive of a 32-bit fractional dividend and a 16-bit fractional divisor, returning a 16-bit result. If both operands are equal, returns 0x7FFF (occurs naturally).

Note — Does not check for division overflow cases.

Does not check for divide by zero cases.

Prototype
```
Word16 div_ls(Word16 s_denominator, Word32
l_numerator)
```

Example
```
long  l =0x20000000;/*  0.25 */
short s2=0x4000;/*  0.5  */
short result;

result = div_ls(s2,s1);
// Expected value of result: 0.25/0.5 = 0.5 =
0x4000
```

div_ls4q

Definition — Four quadrant division of a 32-bit fractional dividend and a 16-bit fractional divisor, returning a 16-bit result.

Note: Does not check for division overflow cases.

Does not check for divide by zero cases.

Prototype

```
Word16 div_ls4q(Word32 l_numerator, Word16
s_denominator)
```

Example

```
long  l =0xE0000000;/*  -0.25 */
short s2=0xC000;/*  -0.5  */
short result;

result = div_ls4q(s1,s2);
// Expected value of result: -0.25/-0.5 = 0.5 =
0x4000
```

Multiplication/ MAC

Click any of the following links to jump to the corresponding intrinsic functions:

- mac_r
- msu_r
- mult
- mult_r
- L_mac
- L_msu
- L_mult
- L_mult_ls

---

mac_r

Definition Multiply two 16-bit fractional values and add to 32-bit fractional value. Round into a 16-bit result, saturating if necessary. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.

Prototype

```
Word16 mac_r(Word32 laccum,  Word16 sinp1, Word16
sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0x4000;/*   0.5 */
short result;
long Acc = 0x0x0000FFFF;

result = mac_r(Acc,s1,s2);
// Expected value of result: 0xE001
```

msu_r

Definition — Multiply two 16-bit fractional values and subtract this product from a 32-bit fractional value. Round into a 16-bit result, saturating if necessary. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.

Prototype

```
Word16 msu_r(Word32 laccum,  Word16 sinp1, Word16
sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0x4000;/*   0.5 */
short result;
long Acc = 0x20000000;

result = msu_r(Acc,s1,s2);
// Expected value of result: 0x4000
```

mult

Definition — Multiply two 16-bit fractional values and truncate into a 16-bit fractional result. Saturates only for the case of 0x8000 x 0x8000. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 mult(Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0x2000;/*  0.25 */
short s2 = 0x2000;/*  0.25 */
short result;

result = mult(s1,s2);
```

```
// Expected value of result: 0.625 = 0x0800
```

mult_r

Definition — Multiply two 16-bit fractional values, round into a 16-bit fractional result. Saturates only for the case of 0x8000 x 0x8000. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.

Prototype

```
Word16 mult_r(Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0x2000;/*   0.25 */
short s2 = 0x2000;/*   0.25 */
short result;

result = mult_r(s1,s2);
// Expected value of result: 0.625 = 0x0800
```

L_mac

Definition — Multiply two 16-bit fractional values and add to 32-bit fractional value, generating a 32-bit result, saturating if necessary.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_mac(Word32 laccum, Word16 sinp1, Word16
sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0x4000;/*   0.5 */
long result, Acc = 0x20000000;/*  0.25 */

result = L_mac(Acc,s1,s2);
```

```
// Expected value of result: 0
```

L_msu

Definition — Multiply two 16-bit fractional values and subtract this product from a 32-bit fractional value, saturating if necessary. Generates a 32-bit result.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_msu(Word32 laccum, Word16 sinp1, Word16
sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0xC000;/* - 0.5 */
long result, Acc = 0;

result = L_msu(Acc,s1,s2);
// Expected value of result: -0.25
```

L_mult

Definition — Multiply two 16-bit fractional values generating a signed 32-bit fractional result. Saturates only for the case of 0x8000 x 0x8000.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_mult(Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0x2000;/*  0.25 */
short s2 = 0x2000;/*  0.25 */
long result;

result = L_mult(s1,s2);
// Expected value of result: 0.625 = 0x08000000
```

L_mult_ls

Definition — Multiply one 32-bit and one-16-bit fractional value, generating a signed 32-bit fractional result. Saturates only for the case of 0x80000000 x 0x8000.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_mult_ls(Word32 linp1, Word16 sinp2)
```

Example

```
long l1 = 0x20000000;/* 0.25 */
short s2 = 0x2000;/* 0.25 */
long result;

result = L_mult(l1,s2);
// Expected value of result: 0.625 = 0x08000000
```

Normalization

Click any of the following links to jump to the corresponding intrinsic functions:

- ffs_s
- norm_s
- ffs_l
- norm_l

---

ffs_s

Definition — Computes the number of left shifts required to normalize a 16-bit value, returning a 16-bit result (finds 1st sign bit). Returns a shift count of 31 for an input of 0x0000.

Note — Does not actually normalize the value!

Also see the intrinsic ffs_s which handles the case where the input == 0x0000 differently.

Prototype

```
Word16 ffs_s(Word16 ssrc)
```

Example

```
short s1 = 0x2000;/* .25 */
short result;

result = ffs_s(s1);
// Expected value of result: 1
```

norm_s

Definition — Computes the number of left shifts required to normalize a 16-bit value, returning a 16-bit result. Returns a shift count of 0 for an input of 0x0000.

Note — Does not actually normalize the value!

This operation is NOT optimal on the DSP56800E because of the case of returning 0 for an input of 0x0000. See the intrinsic ffs_s which is more optimal but generates a different value for the case where the input == 0x0000.

Prototype

```
Word16 norm_s(Word16 ssrc)
```

Example

```
short s1 = 0x2000;/* .25 */
short result;

result = norm_s(s1);
// Expected value of result: 1
```

ffs_l

Definition — Computes the number of left shifts required to normalize a 32-bit value, returning a 16-bit result (finds 1st sign bit). Returns a shift count of 31 for an input of 0x00000000.

Note — Does not actually normalize the value!

Also, see the intrinsic norm_l which handles the case where the input == 0x00000000 differently.

Prototype

```
Word16 ffs_l(Word32 lsrc)
```

Example

```
long ll = 0x20000000;/* .25 */
short result;

result = ffs_l(ll);
// Expected value of result: 1
```

norm_l

Definition — Computes the number of left shifts required to normalize a 32-bit value, returning a 16-bit result. Returns a shift count of 0 for an input of 0x00000000.

Note — Does not actually normalize the value!

This operation is NOT optimal on the DSP56800E because of the case of returning 0 for an input of 0x00000000. See the intrinsic ffs_l which is more optimal but generates a different value for the case where the input == 0x0000.

Prototype

```
Word16 norm_l(Word32 lsrc)
```

Example

```
long ll = 0x20000000;/* .25 */
short result;

result = norm_l(ll);
// Expected value of result: 1
```

Rounding

Click on the following link to jump to the corresponding intrinsic function:

- round

---

round

Definition — Rounds a 32-bit fractional value into a 16-bit result. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions — OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 round(Word32 lvar1)
```

Example

```
long l = 0x12348002;/*if low 16bit = 0xFFFF >
0x8000 then add 1 */
short result;

result = round(l);
// Expected value of result: 0x1235
```

Shifting

Click any of the following links to jump to the corresponding intrinsic functions:

- shl
- shlftNs
- shlfts
- shr
- shr_r
- shrtNs
- L_shl
- L_shlftNs
- L_shlfts
- L_shr
- L_shr_r
- L_shrtNs

shl

Definition — Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note — This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic shlftNs or shlfts which are more optimal.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 shl(Word16 sval2shft, Word16 s_shftamount)
```

Example

```
short result;
short s1 = 0x1234;
short s2 = 1;

result = shl(s1,s2);
// Expected value of result: 0x2468
```

shlftNs

Definition — Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation does not occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note — Ignores upper N-5 bits of s_shftamount except the sign bit (MSB).

If s_shftamount is positive and the value in the lower 5 bits of s_shftamount is greater than 15, the result is 0.

If s_shftamount is negative and the absolute value in the lower 5 bits of s_shftamount is greater than 15, the result is 0 if sval2shft is positive, and 0xFFFF if sval2shft is negative.

Prototype

```
Word16 shlftNs(Word16 sval2shft, Word16
s_shftamount)
```

Example

```
short result;
short s1 = 0x1234;
short s2 = 1;

result = shlftNs(s1,s2);
// Expected value of result: 0x2468
```

shlfts

Definition — Arithmetic left shift of 16-bit value by a specified shift amount. Saturation does occur during a left shift if required. When an accumulator is the destination, zeroes out the LSP portion.

Note — This is not a bidirectional shift.

Assumptions — Assumed s_shftamount is positive.

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 shlfts(Word16 sval2shft, Word16
s_shftamount)
```

Example

```
short result;
short s1 = 0x1234;
short s2 = 3;

result = shlfts(s1,s2);
// Expected value of result: 0x91a0
```

shr

Definition — Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note — This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic shlfts which is more optimal.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 shr(Word16 sval2shft, Word16 s_shftamount)
```

Example

```
short result;
short s1 = 0x2468;
short s2= 1;

result = shr(s1,s2);
// Expected value of result: 0x1234
```

shr_r

Definition — Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. If a right shift is performed, then rounding performed on result. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note — This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic shlfts which is more optimal.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 shr_r(Word16 s_val2shft, Word16
s_shftamount)
```

Example

```
short result;
short s1 = 0x2468;
short s2= 1;

result = shr(s1,s2);
// Expected value of result: 0x1234
```

shrtNs

Definition — Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation does not occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note — Ignores upper N-5 bits of s_shftamount except the sign bit (MSB).

If s_shftamount is positive and the value in the lower 5 bits of s_shftamount is greater than 15, the result is 0 if sval2shft is positive, and 0xFFFF is sval2shft is negative.

If s_shftamount is negative and the absolute value in the lower 5 bits of s_shftamount is greater than 15, the result is 0.

Prototype

```
Word16 shrtNs(Word16 sval2shft, Word16
s_shftamount)
```

Example

```
short result;
short s1 = 0x2468;
short s2= 1;

result = shrtNs(s1,s2);
// Expected value of result: 0x1234
```

L_shl

Definition — Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note — This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic L_shl or L_shlfts which are more optimal.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_shl(Word32 lval2shft, Word16
s_shftamount)
```

Example

```
long result, l = 0x12345678;
short s2 = 1;

result = L_shl(l,s2);
// Expected value of result: 0x2468ACF0
```

L_shlftNs

Definition — Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation does not occur during a left shift.

Note — Ignores upper N-5 bits of s_shftamount except the sign bit (MSB).

Prototype

```
Word32 L_shlftNs(Word32 lval2shft, Word16
s_shftamount)
```

Example

```
long result, l = 0x12345678;
short s2= 1;

result = L_shlftNs(l,s2);
// Expected value of result: 0x2468ACF0
```

L_shlfts

Definition — Arithmetic left shift of 32-bit value by a specified shift amount. Saturation does occur during a left shift if required.

Note — This is not a bidirectional shift.

Assumptions — Assumed s_shftamount is positive.

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_shlfts(Word32 lval2shft, Word16
s_shftamount)
```

Example

```
long result, l = 0x12345678;
short s1 = 3;

result = shlfts(l, s1);
// Expected value of result: 0x91A259E0
```

L_shr

Definition — Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

Note — This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic L_shrtNs which is more optimal.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_shr(Word32 lval2shft, Word16
s_shftamount)
```

Example

```
long result, l = 0x24680000;
short s2= 1;

result = L_shrtNs(l,s2);
// Expected value of result: 0x12340000
```

L_shr_r

Definition — Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. If a right shift is performed, then rounding performed on result. Saturation may occur during a left shift.

Assumptions — OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

Prototype
```
Word32 L_shr_r(Word32 lval2shft, Word16
s_shftamount)
```

Example
```
long l1 = 0x41111111;
short s2 = 1;
long result;

result = L_shr_r(l1,s2);
// Expected value of result: 0x20888889
```

L_shrtNs

Definition — Arithmetic shift of 32-bit value by a specified shift amount.If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation does not occur during a left shift.

Note — Ignores upper N-5 bits of s_shftamount except the sign bit (MSB).

Prototype
```
Word32 L_shrtNs(Word32 lval2shft, Word16
s_shftamount)
```

Example
```
long result, l = 0x24680000;
short s2= 1;

result = L_shrtNs(l,s2);
// Expected value of result: 0x12340000
```

Modulo Addressing Intrinsic Functions

This section includes the following sub-sections:

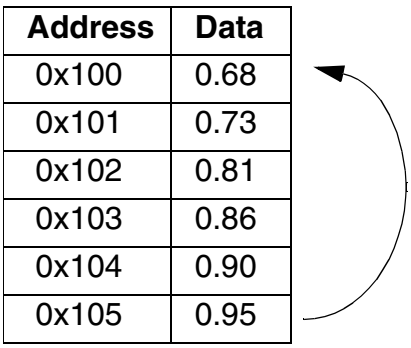
- An Overview of Modulo Addressing
- Modulo Addressing Intrinsic Functions: Definitions and Examples
- Caveats on Using the Modulo Buffer API
- Error Code Documentation

An Overview of Modulo Addressing

A modulo buffer is a region of memory or a buffer in which the pointer to the buffer data loops back to the beginning of the buffer if the pointer exceeds some specified limit.

For example, in Figure 6.2 the address loops to the initial buffer address after the specified modulo buffer limit is reached. For this example, the modulo buffer limit is six.

Figure 6.2 Example of a Modulo Buffer



In order to efficiently implement modulo buffers in C, the DSP56800E C compiler uses intrinsic functions to create and manipulate modulo buffers in C. Normally, a modulo operation, that is, the % operator in C requires a runtime function call to the arithmetic library to perform the binary operation which results in a a large execution time overhead for normally timed critical DSP loops. However, with our implementation of modulo buffers in C, the runtime call to perform the modulo operation has been eliminated and replaced with an efficient implementation of circular

address modification, by either using hardware resources or by directly manipulating the address mathematically.

Since modulo buffers occur often in DSP programming, these buffers have hardware support on-chip. Specifically, modulo control registers work in conjunction with the pointer update addressing modes of the DSP such that a range of addresses is accessed, rather than a continuous, linear address space. But hardware support for modulo buffers have strict requirements on buffer address alignment, pointer register resources (only R0 and R1 update in a modulo fashion), and limited modulo addressing instructions. Because of these hardware limitations, the compiler actually implements modulo buffers by manipulating the buffers via intrinsic or built-in function calls.

In summary, we use a well-defined set of instrinsic APIs to implement modulo buffers in C for the DSP56800E compiler.

Modulo Addressing Intrinsic Functions: Definitions and Examples

This section describes the intrinsic functions used for modulo addressing. These functions are:

- __mod_init
- __mod_initint16
- __mod_start
- __mod_access
- __mod_update
- __mod_stop
- __mod_getint16
- __mod_setint16
- __mod_error

__mod_init

Definition

Initialize a modulo buffer pointer with arbitrary data using the address specified by the <addr_expr>. This function expects a byte address. <addr_expr> is an arbitrary C expression which normally evaluates the address at the beginning of the modulo buffer, although it may be any legal buffer address. The <mod_desc> evaluates to a compile time constant of either 0 or 1, represented by the modulo pointers R0 or R1, respectively. The <mod_sz> is a compile time integer constant representing the size of the modulo buffer in bytes. The <data_sz> is a compile time integer constant representing the size of data being stored in the buffer in bytes. <data_sz> is usually derived from the sizeof() operator.

The __mod_init function may be called independently for each modulo pointer register.

If __mod_error has not been previously called, no record of __mod_init errors are saved.

If __mod_error has been previously called, __mod_init may set one of the error condition in the static memory location defined by __mod_error. (See __mod_error description for a complete list of error conditions).

Prototype

```
void __mod_init (
int <mod_desc>,
void * <addr_expr>,
int <mod_sz>,
int <data_sz>  );
```

Example

Initialize a modulo buffer pointer with a buffer size of 3 and where each element is a structure.

```
__mod_init(0, (void *)&struct_buf[0], 3,
sizeof(struct mystruct) );
```

__mod_initint16

Definition — Initialize modulo buffer pointer with integer data. The __mod_initint16 function behaves similarly to the __mod_init function, except that word addresses are used to initialize the modulo pointer register.

Prototype

```
void __mod_initint16(
int <mod_desc>,
int * <addr_expr>,
int <mod_sz> );
```

Example — Initialize an integer modulo buffer pointer with a buffer size of 10.

```
__mod_initint16(0, &int_buf[9], 10);
```

__mod_start

Definition — Write the modulo control register. The __mod_start function simply writes the modulo control register (M01) for each modulo pointer register which has been previously initialized. The values written to M01 depends on the size of the modulo buffer and which pointers have been initialized.

Prototype

```
void __mod_start( void );
```

__mod_access

Definition — Retrieve the modulo pointer. The __mod_access function returns the modulo pointer value specified by <mod_desc> in the R2 register, as per calling conventions. The value returned is a byte address. The data in the modulo buffer may be read or written by a cast and dereference of the resulting pointer.

Prototype

```
 void  *__mod_access( int <mod_desc>);
```

Example — Assign a value to the modulo buffer at the current pointer.

```
*((char *)__mod_access(0)) = (char)i;
```

__mod_update

Definition — Update the modulo pointer. The __mod_update function updates the modulo pointer by the number of data type units specified in <amount>. <amount> may be negative. Of course, the pointer will wrap to the beginning of the modulo buffer if the pointer is advanced beyond the modulo boundaries. <amount> must be a compile time constant.

Prototype

```
void  __mod_update( int <mod_desc>, int <amount>);
```

Example — Advance the modulo pointer by 2 units.

```
__mod_access(0, 2);
```

__mod_stop

Definition — Reset modulo addressing to linear addressing. This function writes the modulo control register with a value which restore linear addressing to the R0 and R1 pointer registers.

Prototype

```
void  __mod_stop( int <mod_desc );
```

__mod_getint16

Definition — Retrieve a 16-bit signed value from the modulo buffer and update the modulo pointer.This function returns an integer value from the location pointed to by the modulo pointer. The function then updates the modulo pointer by <amount> integer units (<amount>*2 bytes). <amount> must be a compile time constant.

Prototype

```
int __mod_getint16( int <mod_desc>, int <amount>
);
```

Example — Retrieve an integer value from a modulo buffer and update the modulo buffer pointer by one word.

```
int y;
y = __mod_getint16(0, 1);
```

__mod_setint16

Definition — Write a 16-bit signed integer to the modulo buffer and update the pointer. This function evaluates <int_expr> and copies the value to the location pointed to by the modulo pointer. The modulo pointer is then updated by <amount>. <amount> must be a compile time constant.

Prototype

```
int __mod_setint16( int <mod_desc>, int
<int_expr>, int <amount> );
```

Example — Write the modulo buffer with a value derived from an expression, do not update modulo pointer.

```
__mod_setint16( 0, getrandomint(), 0 );
```

__mod_error

Definition — Set up a modulo error variable. This function registers a static integer address to hold the error results from any of the modulo buffer API calls. The function returns 0 if it is successful, 1 otherwise. The argument must be the address of a static, global integer variable. This variable holds the result of calling each of the previously defined API functions. This allows the user to monitor the status of the error variable and take action if the error variable is non-zero. Typically, the user would use __mod_error during development and remove it once debugging is complete. __mod_error generates no code, although the error variable may occupy a word of memory. A non-zero value in the error variable indicates a misuse of the one of the API functions. Once the error variable is set it is reset when __mod_stop is called. The error variable contains the error number of the last error. A successful

call to an API function will not reset the error variable; only __mod_stop will reset the error variable.

Prototype

```
int __mod_error( int * <static_object_addr>);
```

Example

```
Register the error number variable
static int myerrno;

assert( __mod_error(&myerrno) == 0 ) ;
```

Modulo Buffer Examples

In this section, we provide two modulo buffer examples that are shown in Listing 6.12 and Listing 6.13.

Listing 6.12 Modulo Buffer (Example 1)

```
#pragma define_section DATA_INT_MODULO ".data_int_modulo"


/* Place the buffer object in a unique section so the it can be
aligned properly in the linker control file. */

#pragma section DATA_INT_MODULO begin
int int_buf[10];
#pragma section DATA_INT_MODULO end


  /* Convenient defines for modulo descriptors */

#define M0 0
#define M1 1

int main ( void )
{
  int i;

/*   Modulo buffer will be initialized. R0 will be the modulo
pointer register. The buffer size is 10 units. The unit size is
'sizeof(int)'. */

__mod_init(M0, (void *)&int_buf[0], 10, sizeof(int));
```

```
/* Write the modulo control register */

  __mod_start();

/* Write int_buf[0] through int_buf[9]. R0 initially points at
int_buf[0] and wraps when the pointer value exceeds int_buf[9].
The pointer is updated by 1 unit each time through the loop */

  for ( i=0; i<100; i++ )
  {

    *((int *)__mod_access(M0)) = i;
    __mod_update(M0, 1);

  }

/* Reset modulo control register to linear addressing mode */
  __mod_stop();

}
```

Listing 6.13 Modulo Buffer (Example 2)

```
/* Set up a static location to save error codes */
if ( ! __mod_error(&err_codes)) {
  printf ("__mod_error set up failed\n");
}

/* Initialize a modulo buffer pointer, pointing to an array of 10
ints. */

__mod_initint16(M0, &int_buf[9], 10);

/* Check for success of previous call */

      if ( err_code ) { printf ( "__mod_initint16 failed\n" ) };

  __mod_start();
```

```
/* Write modulo buffer with the result of the expression "i".
Decrement the buffer pointer for each execution of the loop.
The modulo buffer wraps from index 0 to 9 through the entire
execution of the loop. */

  for ( i=100; i>0; i-- ) {

    __mod_setint16(M0, i, -1);

  }
  __mod_stop();
```

Caveats on Using the Modulo Buffer API

Modulo buffers must be aligned properly. See the *M56800E User's Manual* on modulo buffer alignment constraints. There is no run-time validation of alignment of modulo buffers. Using the modulo buffer API on unaligned buffers will cause erratic and unpredictable behavior during data accesses.

Writing the modulo control register via a `__mod_start()` call essentially changes the global address generation state of the hardware. Therefore, all user function calls, run-time supporting function calls, standard library calls, and interrupts are affected by this change of state.

NOTE It is the customer's responsibility to make sure that any side-effects (i.e. R0 and R1 update in a modulo way) of enabling the modulo addressing state are accounted for.

Enabling the modulo use of the R1 address register also enables the R0 address register, even if R0 is not explicitly initialized with `__mod_init()` or `__mod_initint16()`. If only one modulo pointer is required, we recommend using R0 only.

A successful API call does not clear the error code in the error variable. Only `__mod_stop` clears the error code in the error variable.

Error Code Documentation

If you want to register a static variable as error code storage use `__mod_error()`. Should an error occur, the error code storage will contain one of the values shown in Table 6.2. For a list of modulo addressing intrinsic functions and their possible error codes see Table 6.3.

Table 6.2 Description of Error Codes for Modulo Addressing

Value	Description
11	<mod_desc> parameter must be zero or one.
12	R0 modulo pointer is already initialized. An extraneous call to `__mod_init` or `__mod_initint16` to initialize R0 has been made.
13	R1 modulo pointer is already initialized. An extraneous call to `__mod_init` or `__mod_initint16` to initialize R1 has been made.
14	Modulo buffer size must be a compile time constant.
15	Modulo buffer size must be greater than one.
16	Modulo buffer size is too big.
17	Modulo buffer size for R0 and R1 must be the same.
18	Modulo buffer data types for R0 and R1 must be the same.
19	Modulo buffer has not been initialized.
20	Modulo buffer has not been started.
21	Parameter is not a compile time constant.
22	Attempt to use word pointer functions with byte pointer initialization. `__mod_getint16` and `__mod_setint16` were called but `__mod_init` was used for initialization. `__mod_initint16` is required for pointer initialization.

Value	Description
23	Modulo increment value is too big for modulo buffer size. The increment value must be less than or equal to the buffer size.
24	Illegal use of R1 as a modulo pointer. R0 must be initialized as modulo before R1 can be used in a modulo context.

Table 6.3 Description of Error Codes for Modulo Addressing

Function	Possible Error Code
`__mod_init`	11, 12, 13, 14, 15, 16, 17, 18, 21
`__mod_stop`	none
`__mod_getint16`	11, 14, 22, 24
`__mod_setint16`	11, 14, 22, 24
`__mod_start`	none
`__mod_access`	11, 19, 24
`__mod_update`	11, 14, 23, 24
`__mod_initint16`	11, 12, 13, 14, 15, 16, 17

7

Debugging for DSP56800E

This chapter describes the generic features of the CodeWarrior™ debugger.

This chapter contains the following sections:

- Target Settings for Debugging
- Command Converter Server
- Load/Save Memory
- Fill Memory
- Save/Restore Registers
- EOnCE Debugger Features
- Using the DSP56800E Simulator
- Launching and Operating the Debugger
- Register Details Window
- Loading a .elf File without a Project
- Command-Line Debugging
- System-Level Connect
- Debugging in the Flash Memory
- Notes for Debugging on Hardware

Target Settings for Debugging

This section explains how to control the debugger by modifying the appropriate settings panels.

To properly debug DSP56800E software, you must set certain preferences in the **Target Settings** window. The **M56800E Target** panel is specific to DSP56800E development. The remaining settings panels are generic to all build targets.

Other settings panels can affect debugging. Table 7.1 lists these panels.

Table 7.1 Setting Panels that Affect Debugging

This panel…	Affects…	Refer to…
M56800E Linker	symbolics, linker warnings	"Linker Issues for DSP56800E" on page 109
M56800E Processor	optimizations	"Optimizing Code for DSP56800E" on page 95
Debugger Settings	Debugging options	Table 4.2 on page 40
Remote Debugging	Debugging communication protocol	"Remote Debugging" on page 70

The **M56800E Target** panel is unique to DSP56800E debugging. The available options in this panel depend on the DSP56800E hardware you are using and are described in detail in the section on "Remote Debug Options" on page 72.

Command Converter Server

The command converter server (CCS) handles communication between the CodeWarrior debugger and the target board. An icon in the status bar indicates the CCS is running. The CCS is automatically launched by your project when you start a CCS debug session if you are debugging a target board using a local machine. However, when debugging a target board connected to a remote machine, see "Setting Up a Remote Connection" on page 174.

NOTE Projects are set to debug locally by default. The protocol the debugger uses to communicate with the target board, for example, PCI, is determined by how you installed the CodeWarrior software. To modify the protocol, make changes in the **Metrowerks Command Converter Server** window (Figure 7.3).

Essential Target Settings for Command Converter Server

Before you can download programs to a target board for debugging, you must specify the target settings for the command converter server:

- **Local Settings**

  If you specify that the CodeWarrior IDE start the command converter server locally, the command converter server uses the connection port (for example, LPT1) that you specified when you installed CodeWarrior for DSP56800E.

- **Remote Settings**

  If you specify that the CodeWarrior IDE start the command converter server on a remote machine, specify the IP address of the remote machine on your network (as described in "Setting Up a Remote Connection" on page 174.)

- **Default Settings**

  By default, the command converter server listens on port 41475. You can specify a different port number for the debugger to connect to if needed (as described in "Setting Up a Remote Connection" on page 174.) This is necessary if the CCS is configured to a port other than 41475.

After you have specified the correct settings for the command converter server (or verified that the default settings are correct), you can download programs to a target board for debugging.

The CodeWarrior IDE starts the command converter server at the appropriate time if you are debugging on a local target.

Before debugging on a board connected to a remote machine, ensure the following:

- The command converter server is running on the remote host machine.
- Nobody is debugging the board connected to the remote host machine.

Changing the Command Converter Server Protocol to Parallel Port

If you specified the wrong parallel port for the command converter server when you installed CodeWarrior for DSP56800E, you can change the port.

Change the parallel port:

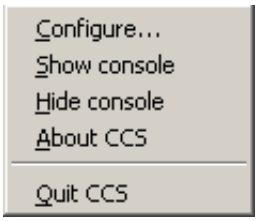
1. **Click the command converter server icon.**

   While the command converter server is running, locate the command converter server icon on the status bar. Right-click on the command converter server icon (Figure 7.1):

Figure 7.1 Command Converter Server Icon

A menu appears (Figure 7.2):

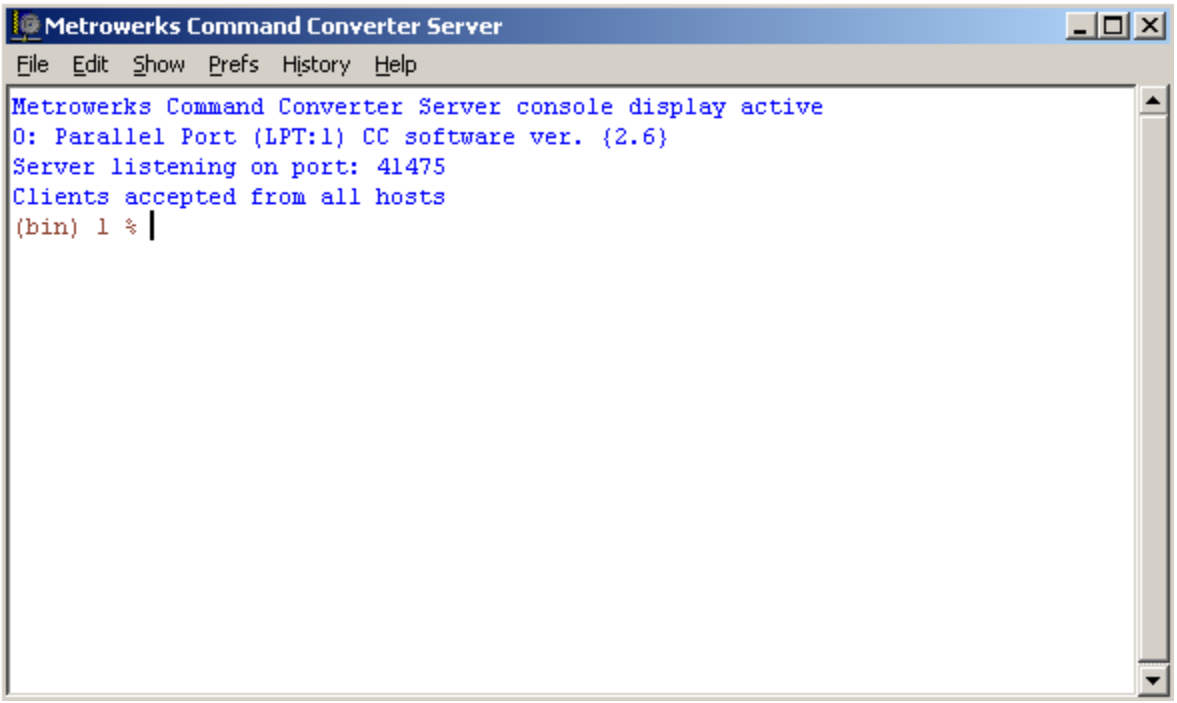
Figure 7.2 Command Converter Server Menu



2. **Select Show console from the menu.**

   The **Metrowerks Command Converter Server** window appears (Figure 7.3).

Figure 7.3 Metrowerks Command Converter Server Window


Metrowerks Command Converter Server
File Edit Show Prefs History Help
Metrowerks Command Converter Server console display active
0: Parallel Port (LPT:1) CC software ver. (2.6)
Server listening on port: 41475
Clients accepted from all hosts
(bin) 1 %

3. **On the console command line, type the following command:**

   ```
   delete all
   ```

4. **Press Enter.**

5. **Type the following command, substituting the number of the parallel port to use (for example, 1 for LPT1):**

   ```
   config cc parallel:1
   ```

6. **Press Enter.**

7. **Type the following command to save the configuration:**

   ```
   config save
   ```

8. **Press Enter.**

Changing the Command Converter Server Protocol to PCI

To change the command converter server to a PCI Connection:

1. **While the command converter server is running, right-click on the command converter server icon shown in Figure 7.1 or double click on it.**
2. **From the menu shown in Figure 7.2, select Show Console.**
3. **At the console command line in the Metrowerks Command Converter Server window shown in Figure 7.3, type the following command:**

   ```
   delete all
   ```

4. **Press Enter.**
5. **Type the following command:**

   ```
   config cc pci
   ```

6. **Press Enter.**
7. **Type the following command to save the configuration:**

   ```
   config save
   ```

8. **Press Enter.**

Setting Up a Remote Connection

A remote connection is a type of connection to use for debugging along with any preferences that connection may need. To change the preferences for a remote connection or to create a new remote connection:

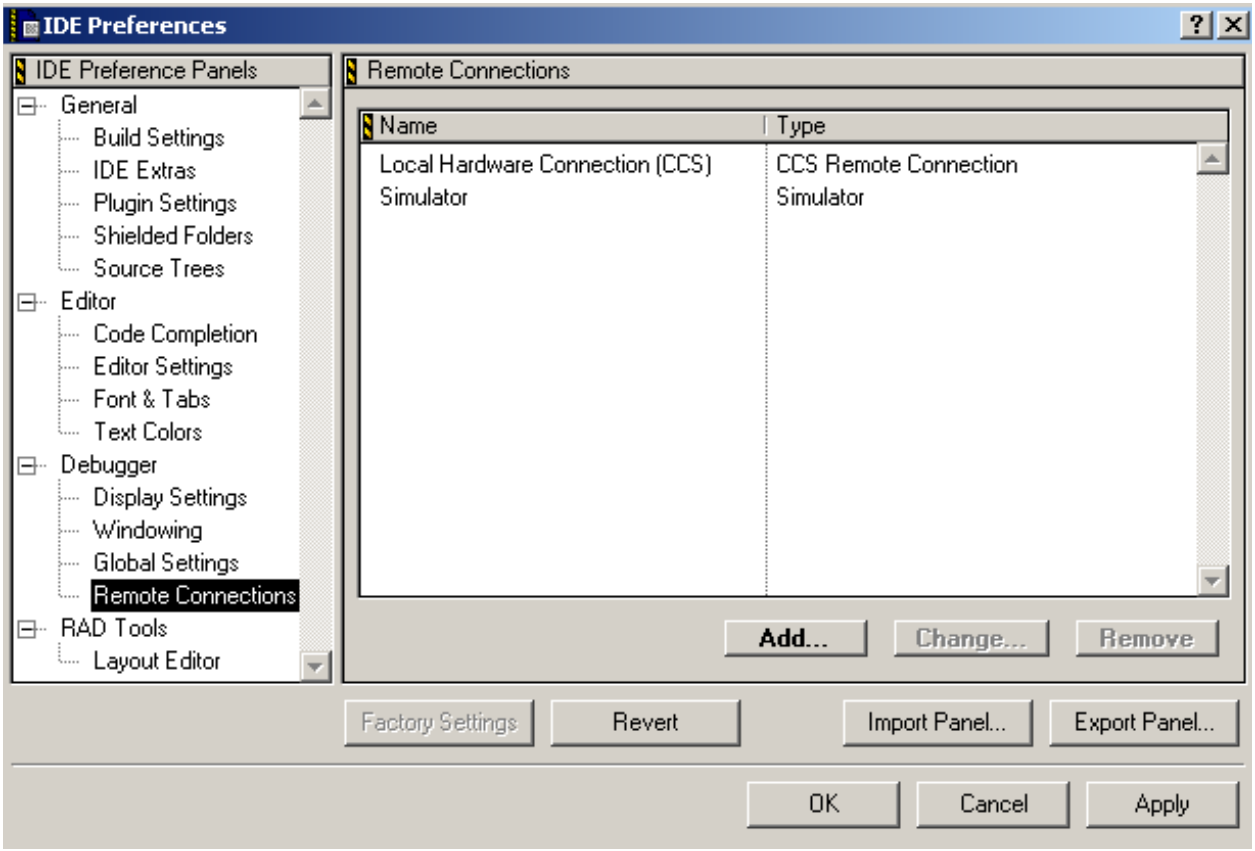
1. **On the main menu, select Edit > Preferences.**

   The IDE Preferences Window appears.

2. **Click Remote Connections in the left column.**

   The **Remote Connections** panel shown in Figure 7.4 appears.

Figure 7.4 Remote Connections Panel



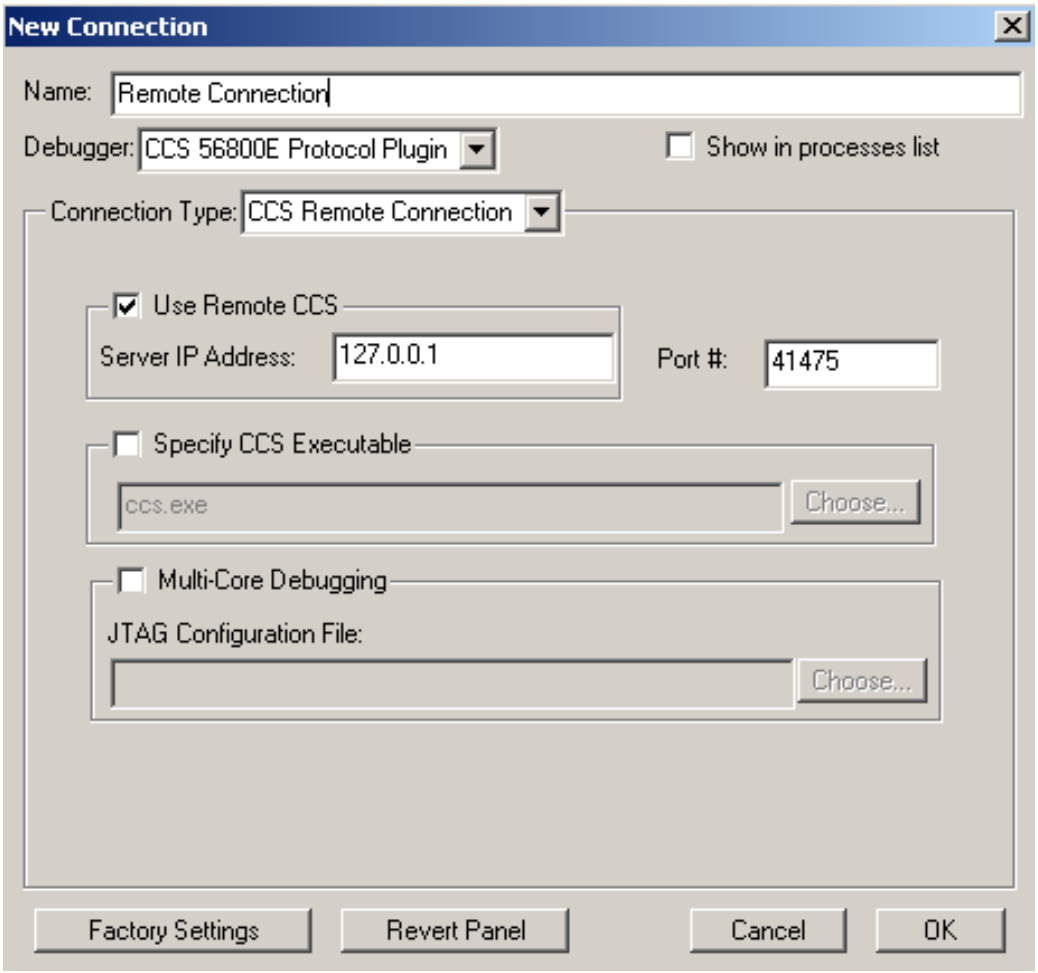
To Add a New Remote Connection

To add a new remote connection:

1. **Click the Add button.**

   The **New Connection** window appears as shown in Figure 7.5.

Figure 7.5 New Connection Window



2. **In the Name edit box, type in the connection name.**
3. **Check Use Remote CCS checkbox.**

   Select this checkbox to specify that the CodeWarrior IDE is connected to a remote command converter server. Otherwise, the IDE starts the command converter server locally
4. **Enter the Server IP address or host machine name.**

   Use this text box to specify the IP address where the command converter server resides when running the command converter server from a location on the network.
5. **Enter the Port # to which the command converter server listens or use the default port, which is 41475.**

6. **Click the OK button.**

To Change an Existing Remote Connection

To change an existing remote connection:

Double click on the connection name that you want to change, or click once on the connection name and click the **Change** button (shown in Figure 7.4 in grey).

To Remove an Existing Remote Connection

To remove an existing remote connection:

Click once on the connection name and click the **Remove** button (shown in Figure 7.4 in grey).

Debugging a Remote Target Board

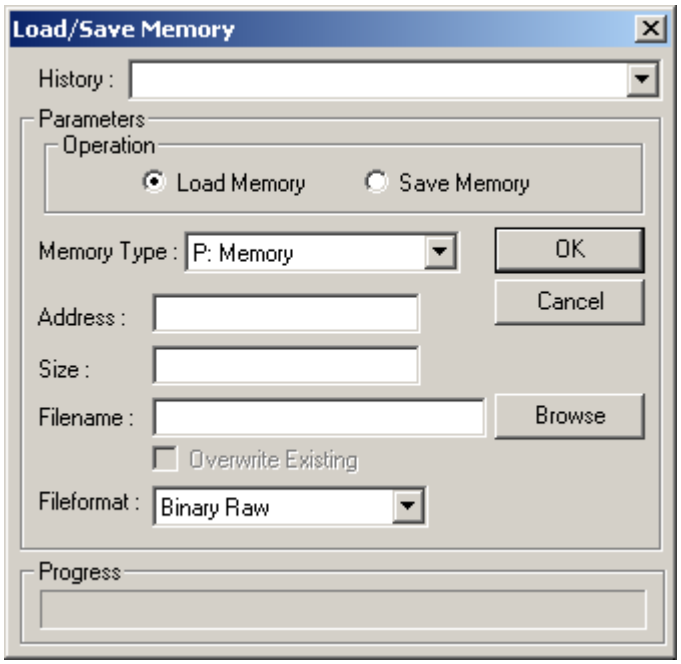
For debugging a target board connected to a remote machine with Code Warrior IDE installed, perform the following steps:

1. **Connect the target board to the remote machine.**
2. **Launch the command converter server (CCS) on the remote machine with the local settings configuration using instructions described in the section "Essential Target Settings for Command Converter Server" on page 171.**
3. **In the Target Settings>Remote Debugging panel for your project, make sure the proper remote connection is selected.**
4. **Launch the debugger.**

Load/Save Memory

From the menu bar of the Metrowerks CodeWarrior window, select **Debug > Load/Save Memory** to display the **Load/Save Memory** dialog box (Figure 7.6).

Figure 7.6 Load/Save Memory Dialog Box



Use this dialog box to load and save memory at a specified location and size with a user-specified file. You can associate a key binding with this dialog box for quick access. Press the **Tab** key to cycle through the dialog box displays, which lets you quickly make changes without using the mouse.

History Combo Box

The **History** combo box displays a list of recent loads and saves. If this is the first time you load or save, the **History** combo box is empty. If you load/save more than once, the combo box fills with the memory address of the start of the load or save and the size of the fill, to a maximum of ten sessions.

If you enter information for an item that already exists in the history list, that item moves up to the top of the list. If you perform another operation, that item appears first.

NOTE By default, the **History** combo box displays the most recent settings on subsequent viewings.

Radio Buttons

The **Load/Save Memory** dialog box has two radio buttons:

- **Load Memory**
- **Save Memory**

The default is **Load Memory**.

Memory Type Combo Box

The memory types that appear in the **Memory Type** Combo box are:

- P: Memory (Program Memory)
- X: Memory (Data Memory)

Address Text Field

Specify the address where you want to write the memory. If you want your entry to be interpreted as hex, prefix it with `0x`; otherwise, it is interpreted as decimal.

Size Text Field

Specify the number of words to write to the target. If you want your entry to be interpreted as hex, prefix it with `0x`; otherwise, it is interpreted as decimal.

Dialog Box Controls

Cancel, Esc, and OK

In Load and Save operations, all controls are disabled except **Cancel** for the duration of the load or save. The status field is updated with the current progress of the operation. Clicking **Cancel** halts the operation, and re-enables the controls on the dialog box. Clicking **Cancel** again closes the dialog box. Pressing the **Esc** key is same as clicking the **Cancel** button.

With the **Load Memory** radio button selected, clicking **OK** loads the memory from the specified file and writes it to memory until the end of the file or the size specified is reached. If the file does not exist, an error message appears.

With the **Save Memory** radio button selected, clicking **OK** reads the memory from the target piece by piece and writes it to the specified file. The status field is updated with the current progress of the operation.

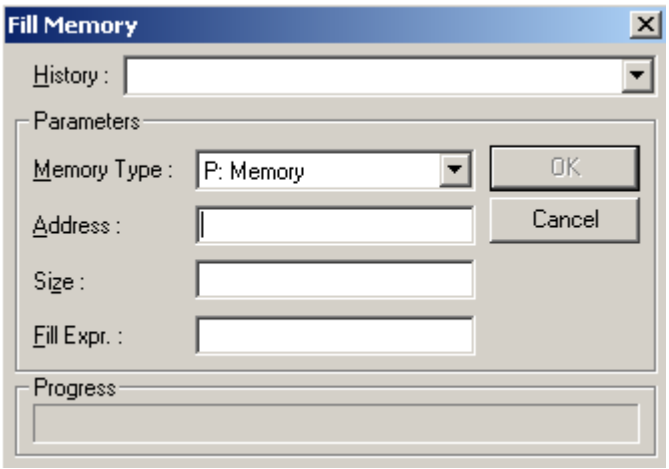
Browse Button

Clicking the **Browse** button displays OPENFILENAME or SAVEFILENAME, depending on whether you selected the **Load Memory** or **Save Memory** radio button.

Fill Memory

From the menu bar of the Metrowerks CodeWarrior window, select **Debug > Fill memory** to display the **Fill Memory** dialog box (Figure 7.7).

Figure 7.7 Fill Memory Dialog Box



Use this dialog box to fill memory at a specified location and size with user- specified raw memory data. You can associate a key binding with this dialog box for quick access. Press the **Tab** key to cycle through the dialog box display, which lets you quickly make changes without using the mouse.

History Combo Box

The **History** combo box displays a list of recent fill operations. If this is the first time you perform a fill operation, the **History** combo box is

empty. If you do more than one fill, then the combo box populates with the memory address of that fill, to a maximum of ten sessions.

If you enter information for an item that already exists in the history list, that item moves up to the top of the list. If you do another fill, then this item is the first one that appears.

NOTE By default, the **History** combo box displays the most recent settings on subsequent viewings.

Memory Type Combo Box

The memory types that can appear in the **Memory Type** Combo box are:

- P:Memory (Program Memory)
- X:Memory (Data Memory)

Address Text Field

Specify the address where you want to write the memory. If you want it to be interpreted as hex, prefix it with `0x`; otherwise, it is interpreted as decimal.

Size Text Field

Specify the number of words to write to the target. If you want it to be interpreted as hex, prefix your entry with `0x`; otherwise, it is interpreted as decimal.

Fill Expression Text Field

Fill writes a set of characters to a location specified by the address field on the target, repeatedly copying the characters until the user-supplied fill size has been reached. **Size** is the total words written, not the number of times to write the string.

Interpretation of the Fill Expression

The fill string is interpreted differently depending on how it is entered in the Fill String field. Any words prefixed with `0x` is interpreted as hex bytes. Thus, `0xBE 0xEF` would actually write `0xBEEF` on the target. Optionally, the string could have been set to

`0xBEEF` and this would do the same thing. Integers are interpreted so that the equivalent signed integer is written to the target.

ASCII Strings

ASCII strings can be quoted to have literal interpretation of spaces inside the quotes. Otherwise, spaces in the string are ignored. Note that if the ASCII strings are not quoted and they are numbers, it is possible to create illegal numbers. If the number is illegal, an error message is displayed.

Dialog Box Controls

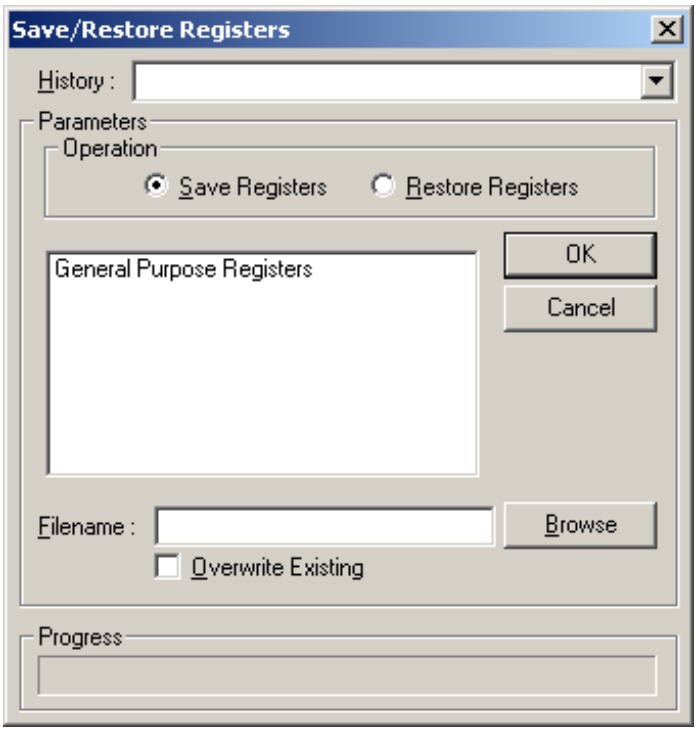
OK, Cancel, and Esc

Clicking **OK** writes the memory piece by piece until the target memory is filled in. The **Status** field is updated with the current progress of the operation. When this is in progress, the entire dialog box grays out except the **Cancel** button, so the user cannot change any information. Clicking the **Cancel** button halts the fill operation, and re-enables the controls on the dialog box. Clicking the **Cancel** button again closes the dialog box. Pressing the **Esc** key is same as pressing the **Cancel** button.

Save/Restore Registers

From the menu bar of the Metrowerks CodeWarrior window, select **Debug > Save/Restore Registers** to display the **Save/Restore Registers** dialog box (Figure 7.8).

Figure 7.8 Save/Restore Registers Dialog Box



Use this dialog box to save and restore register groups to and from a user-specified file.

History Combo Box

The **History** combo box displays a list of recent saves and restores. If this is the first time you have saved or restored, the **History** combo box is empty. If you saved or restored before, the combo box remembers your last ten sessions. The most recent session will appear at the top of the list.

Radio Buttons

The **Save/Restore Registers** dialog box has two radio buttons:

- **Save Registers**
- **Restore Registers**

The default is **Save Registers**.

Register Group List

This list is only available when you have selected **Save Registers**. If you have selected **Restore Registers**, the items in the list are greyed out. Select the register group that you wish to save.

Dialog Box Controls

Cancel, Esc, and OK

In Save and Restore operations, all controls are disabled except **Cancel** for the duration of the load or save. The status field is updated with the current progress of the operation. Clicking **Cancel** halts the operation, and re-enables the controls on the dialog box. Clicking **Cancel** again closes the dialog box. Pressing the **Esc** key is same as clicking the **Cancel** button.

With the **Restore Registers** radio button selected, clicking **OK** restores the registers from the specified file and writes it to the registers until the end of the file or the size specified is reached. If the file does not exist, an error message appears.

With the **Save Register** radio button selected, clicking **OK** reads the registers from the target piece by piece and writes it to the specified file. The status field is updated with the current progress of the operation.

Browse Button

Clicking the **Browse** button displays OPENFILENAME or SAVEFILENAME, depending on whether you selected the **Restore Registers** or **Save Registers** radio button.

EOnCE Debugger Features

The following EOnCE Debugger features are discussed in this section:

- Set Hardware Breakpoint Panel
- Special Counters
- Trace Buffer
- Set Trigger Panel

NOTE These features are only available when debugging with a hardware target.

For more information on the debugging capabilities of the EOnCE, see the EOnCE chapter of your processor's user manual.

Set Hardware Breakpoint Panel

The **Set Hardware BreakPoint** panel (Figure 7.9) lets you set a trigger to do one of the following: halt the processor, cause an interrupt, or start or stop trace buffer capture.

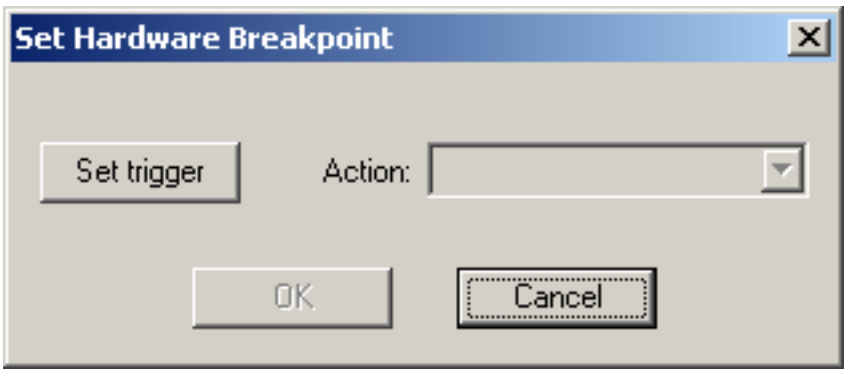
To open this panel:

1. **From the menu bar, select EOnCE > Set Breakpoint Trigger(s).**

To clear triggers set with this panel:

1. **From the menu bar, select EOnCE > Clear Triggers.**

Figure 7.9 Set Hardware Breakpoint Panel



The **Set Hardware BreakPoint** panel options are:

- **Set trigger**

Select this button to open the **Set Trigger** panel (Figure 7.13). For more information on using this panel, see "Set Trigger Panel" on page 190.

- **Action**

This pull down list lets you select the resulting action caused by the trigger.

– **Halt core**

Stops the processor.

– **Interrupt**

Causes an interrupt and uses the vector for the EOnCE hardware breakpoint (unit 0).

Special Counters

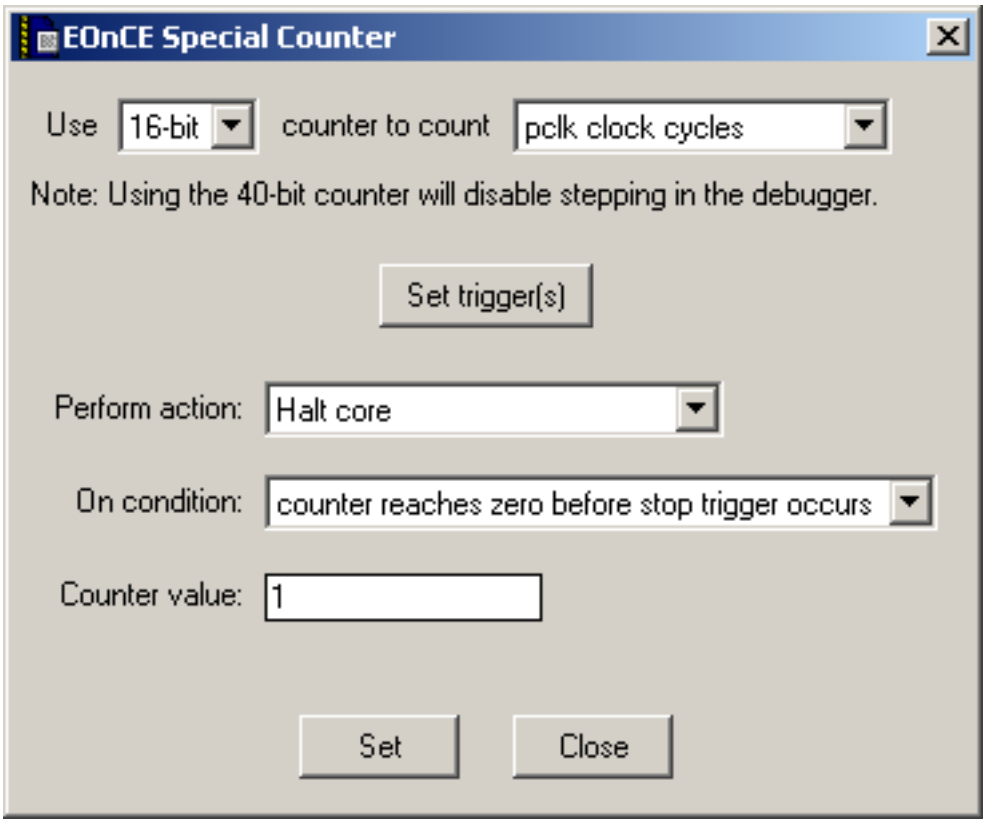
This feature lets you use the special counting function of the EOnCE unit.

To open the EOnCE Special Counter panel (Figure 7.10):

1. **From the menu bar, select EOnCE > Special Counter.**

   This panel is non-modal and will update itself whenever the processor stops.

Figure 7.10 EOnCE Special Counter Panel



The **EOnCE Special Counter** panel options are:

- Counter size

  This pull down list gives you the option to use a 16 or 40-bit counter.

NOTE Using the 40-bit counter will disable stepping in the debugger.

- Counter function

  This pull down list allows you to choose which counting function to use.

- **Set trigger(s)**

  Pushing this button opens the **Set Trigger** panel. For more information on using this panel, see "Set Trigger Panel" on page 190..

- **Perform action**

  This pull down list lets you select the action that occurs when the correct conditions are met, as set in the **Set Trigger** panel and the **On condition** pull down list.

- **On condition**

  This pull down list lets you set the order in which a trigger and counter reaching zero must occur to perform the action specified in **Perform action**.

- **Counter value**

  This edit box should be preloaded with a non-zero counter value when setting the counter. The counter will proceed backward until a stop condition occurs. The edit box will contain the value of the counter and will be updated whenever the processor stops.

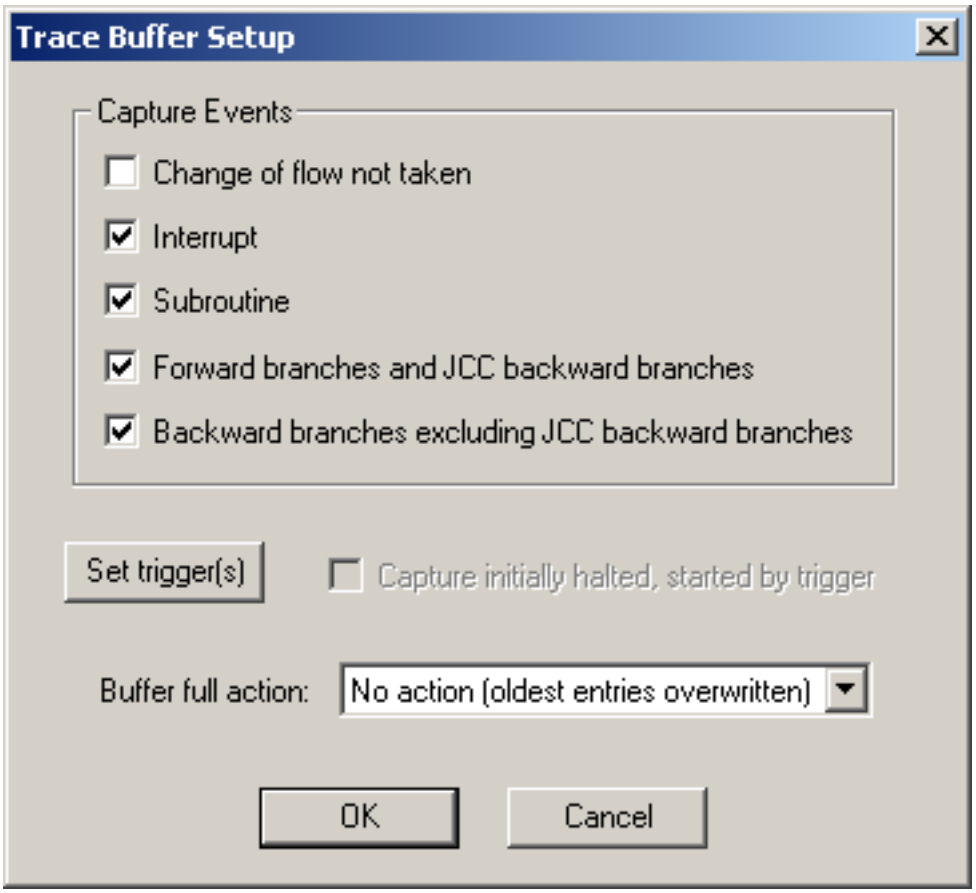
Trace Buffer

The trace buffer lets you view the target addresses of change-of-flow instructions that the program executes. The trace buffer is configured with the **Trace Buffer Setup** panel (Figure 7.11).

To open this panel:

1. **From the menu bar, select Trace Buffer > Setup Trace Buffer.**

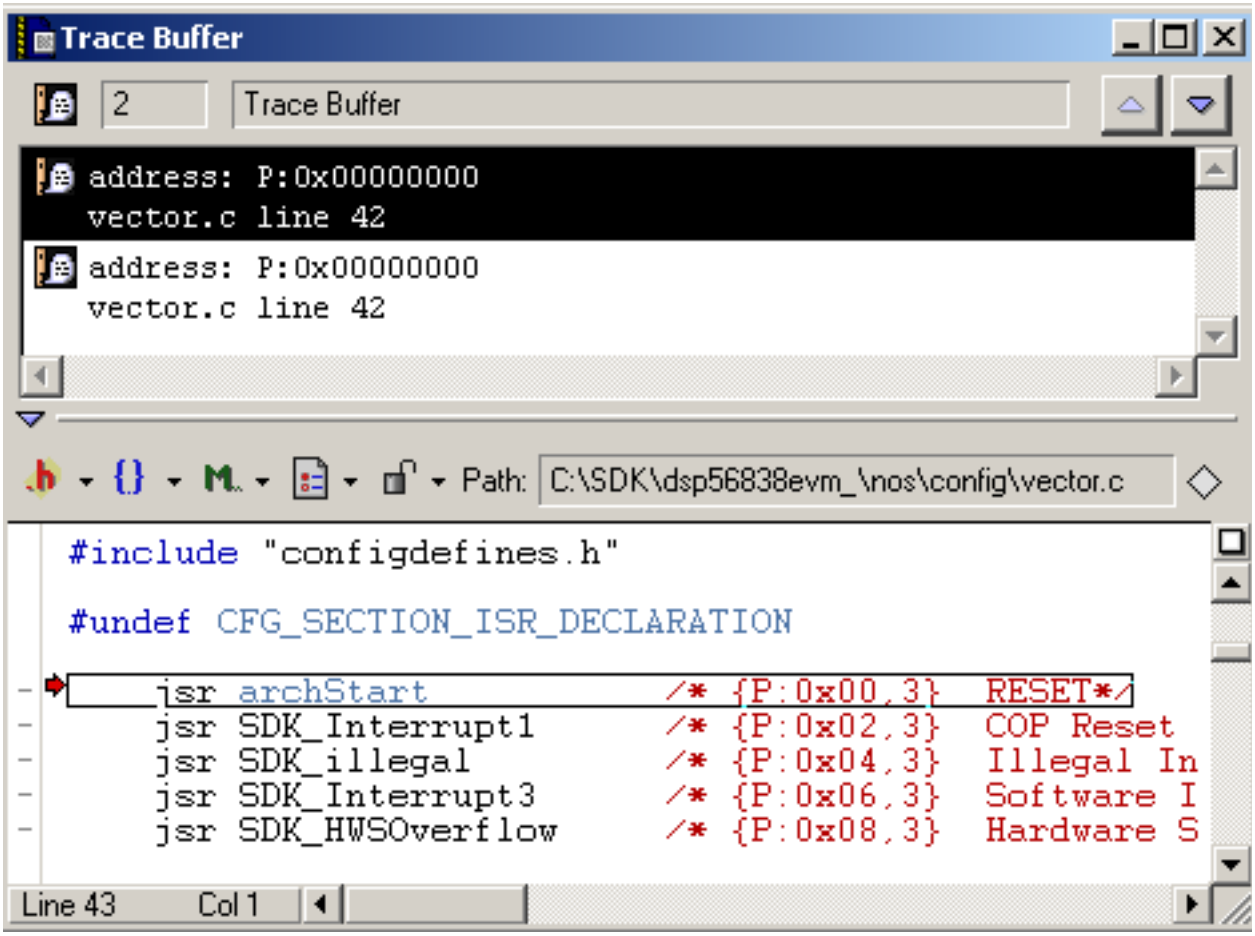
Figure 7.11 Trace Buffer Setup Panel



To view the contents of the trace buffer (Figure 7.12):

1. **From the menu bar, select Trace Buffer > Dump Trace Buffer.**

Figure 7.12 Contents of Trace Buffer



To clear triggers set with the **Trace Buffer Setup** panel (Figure 7.11):

1. **From the menu bar, select EOnCE > Clear Triggers.**

The **Trace Buffer Setup** panel options are:

- **Capture Events**

  Select this set of checkboxes to specify which instructions get captured by the trace buffer.

  – **Change of flow not taken**

    Select this checkbox to capture target addresses of conditional branches and jumps that are not taken.

  – **Interrupt**

    Select this checkbox to capture addresses of interrupt vector fetches and target addresses of RTI instructions.

– **Subroutine**

Select this checkbox to capture target addresses of JSR, BSR, and RTS instructions.

– **Forward branches and JCC Backward branches**

Select this checkbox to capture target addresses of the following taken instructions:

```
BCC forward branch

BRSET forward branch

BRCLR forward branch

JCC forward and backward branches
```

– **Backward branches excluding JCC backward branches**

Select this checkbox to capture target addresses of the following taken instructions:

```
BCC backward branch

BRSET backward branch

BRCLR backward branch
```

- **Set trigger(s)**

  Select this button to open the **Set Trigger** panel (Figure 7.13). For more information on using this panel, see "Set Trigger Panel" on page 191.. The resulting trigger halts trace buffer capture.

- **Capture initially halted, started by trigger**

  When this option is checked, the trace buffer starts off halted.

- **Buffer full action**

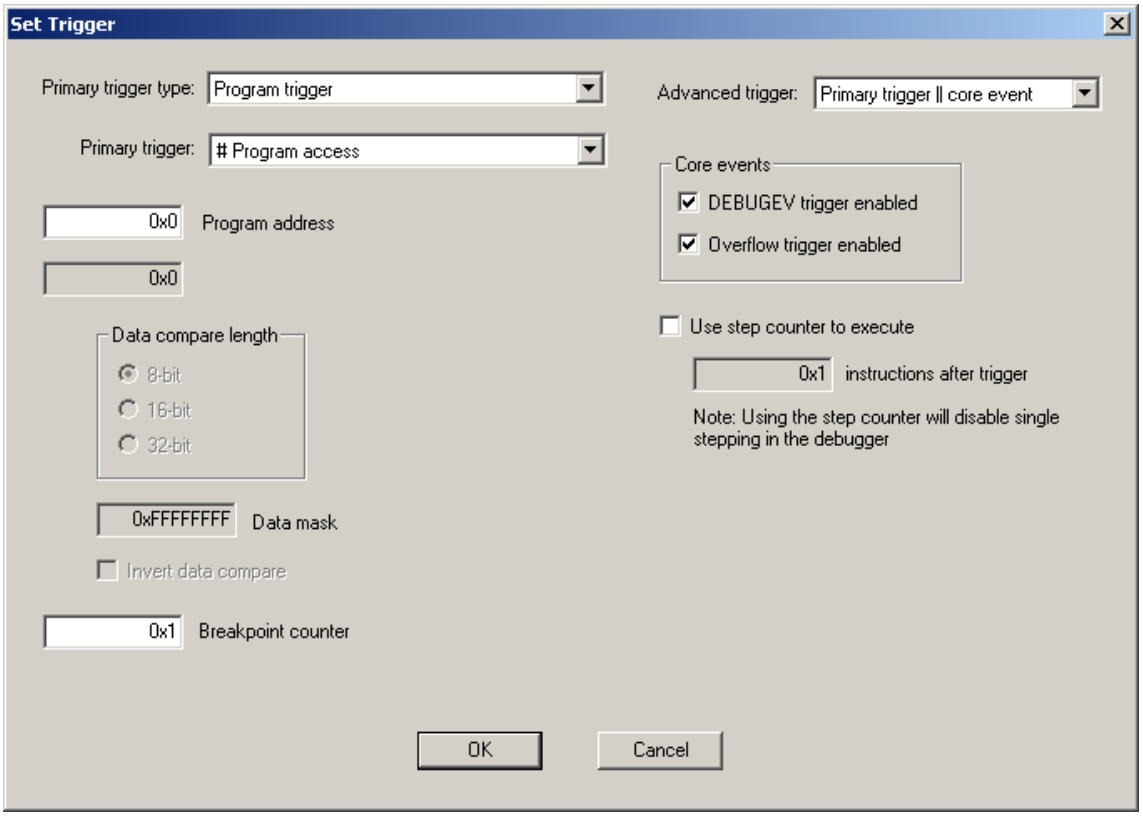
  This pull down list lets you select the resulting action caused by the trace buffer filling.

Set Trigger Panel

The **Set Trigger** panel (Figure 7.13) lets you set triggers for all the EOnCE functions. It can be accessed from the panels used to

configure those functions. The options available change depending on the function being configured.

Figure 7.13 Set Trigger Panel



The **Set Trigger** panel options are:

- **Primary trigger type**

  This pull down list contains the general categories of triggers that can be set.

- **Primary trigger**

  This pull down list contains the specific forms of the triggers that can be set. This list changes depending on the selection made in the **Primary trigger type** option. The # symbol contained in some of the triggers' descriptions specifies that the sub-trigger that it precedes must occur the number of times specified in the **Breakpoint counter** option to cause a trigger. The -> symbol specifies that the first sub-trigger must occur, then the second sub-trigger must occur to cause a trigger.

- **Value options**

  There are two edit boxes used to specify addresses and data values. The descriptions next to the boxes change according to the selection in **Primary trigger type** and **Primary trigger**. According to these options, only one value may be available.

- **Data compare length**

  When the data trigger (address and data) compare trigger is selected, this set of radio buttons becomes available. These options allow you to specify the length of data being compared at that address.

- **Data mask**

  When a data compare trigger is selected, this edit box becomes available. This value specifies which bits of the data value are compared.

- **Invert data compare**

  When a data compare trigger is selected, this checkbox becomes available. When checked, the comparison result of the data value is inverted (logical NOT).

- **Breakpoint counter**

  This edit box specifies the number of times a sub-trigger preceded by a # (see above) must occur to cause a trigger.

- **Advanced trigger**

  This pull down list contains options for combining triggers. The types of triggers that can be combined are triggers set in this panel and core events.

- **Core events**

  This set of checkboxes specify which core events are allowed to enter the breakpoint logic and cause a trigger.

  – **DEBUGEV trigger enabled**

    When this checkbox is selected, the **DEBUGEV** instruction causes a core event.

  – **Overflow trigger enabled**

    When this checkbox is selected, overflow and saturation conditions in the processor cause core events.

- **Use step counter to execute**

When this checkbox is selected, the processor steps through additional instructions after a trigger is signalled. The number of instructions to be stepped is specified in the edit box that is enabled when this checkbox is checked.

Using the DSP56800E Simulator

The CodeWarrior for DSP56800E includes the Motorola DSP56800E Simulator. This software lets you run and debug code on a simulated DSP56800E architecture without installing any additional hardware.

The simulator simulates the DSP56800E processor, not the peripherals. In order to use the simulator, you must select a connection that uses the simulator as your debugging protocol from the **Remote Debugging** panel.

NOTE The simulator also enables the DSP56800E menu for retrieving the machine cycle count and machine instruction count when debugging.

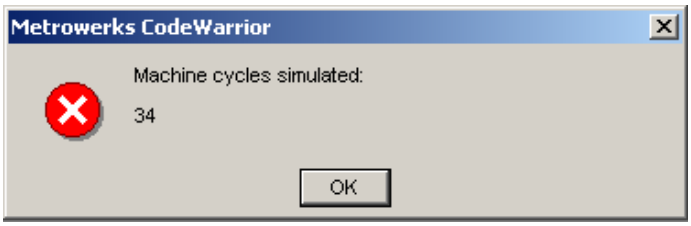
Cycle/Instruction Count

Using the simulator, you can retrieve the machine cycle count and machine instruction count.

Machine Cycle Count

From the menu bar of the Metrowerks CodeWarrior window, select **DSP56800E > Get machine cycle count**. The following window appears (Figure 7.14):

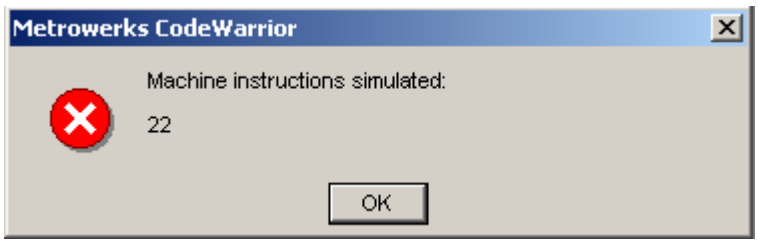
Figure 7.14 Machine Cycle Count



Machine Instruction Count

From the menu bar of the Metrowerks CodeWarrior window, select **DSP56800E > Get machine instruction count**. The following window appears (Figure 7.15):

Figure 7.15 Machine Instruction Count



NOTE Cycle counting is not accurate while single stepping through source code in the debugger. It is only accurate while running. Thus, the cycle counter is more of a profiling tool than an interactive tool.

Launching and Operating the Debugger

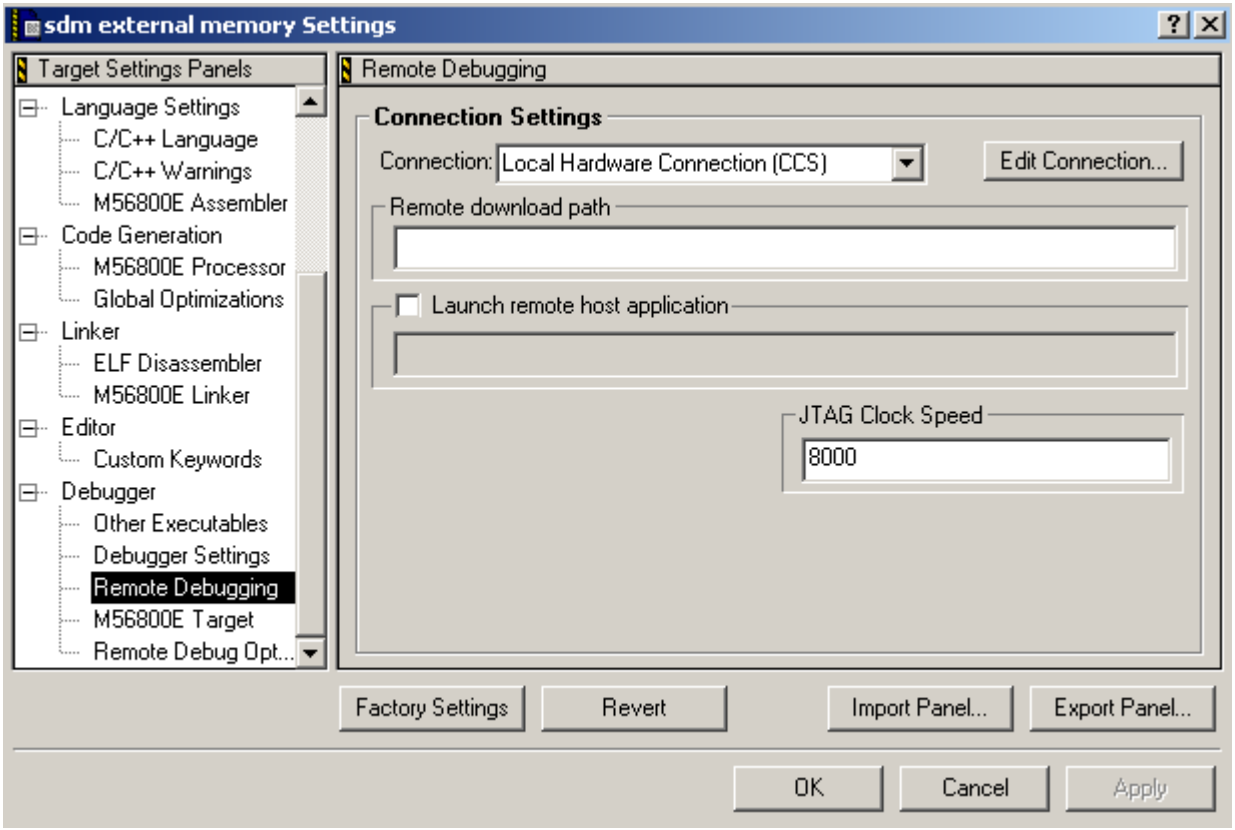
NOTE CodeWarrior IDE automatically enables the debugger and sets debugger-related settings within the project.

1. **Set debugger preferences.**

   Select **Edit >sdm Settings** from the menu bar of the Metrowerks CodeWarrior window.

   The IDE displays the **Remote Debugging** window.

Figure 7.16 Remote Debugging Panel



2. **Select the Connection.**

   For example, select **Local Hardware Connection (CCS).**

3. **Click OK button.**

4. **Debug the project.**

   Use either of the following options:

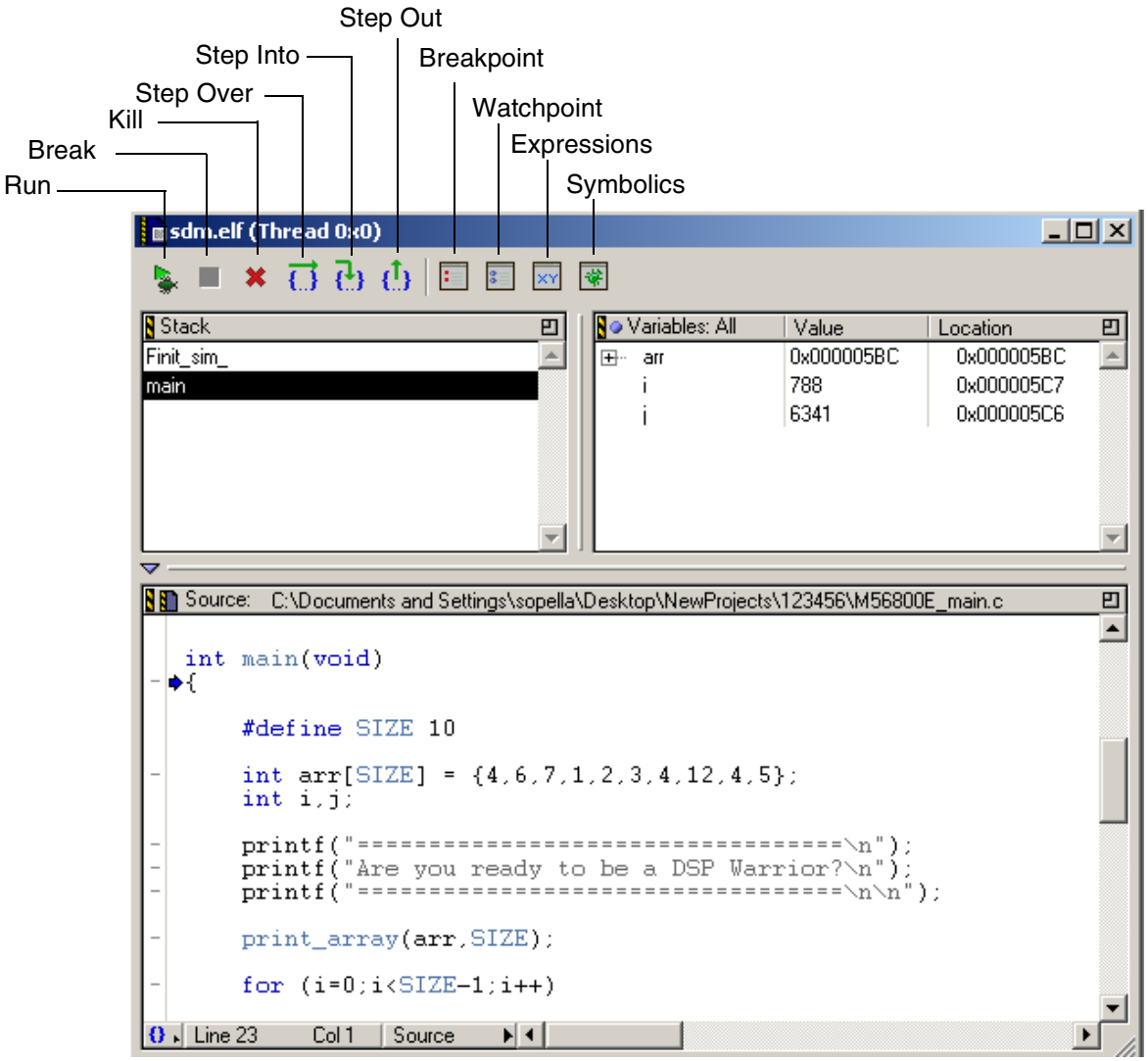
   - From the Metrowerks CodeWarrior window, select **Project > Debug**.
   - Click the **Debug** button in the project window.

   This command resets the board (if **Always reset on download** is checked in the Debugger's **M56800E Target** panel shown in Figure 4.13) and the download process begins.

   When the download to the board is complete, the IDE displays the **Program** window (**sdm.elf** in sample) shown in Figure 7.17.

NOTE Source code is shown only for files that are in the project folder or that have been added to the project in the project manager, and for which the IDE has created debug information. You must navigate the file system in order to locate sources that are outside the project folder and not in the project manager, such as library source files.

Figure 7.17 Program Window



5. **Navigate through your code.**

   The **Program** window has three panes:

   - Stack pane

The **Stack** pane shows the function calling stack.

- Variables pane

  The **Variables** pane displays local variables.

- Source pane

  The **Source** pane displays source or assembly code.

  The toolbar at the top of the window has buttons that allows you access to the execution commands in the **Debug** menu.

Setting Breakpoints

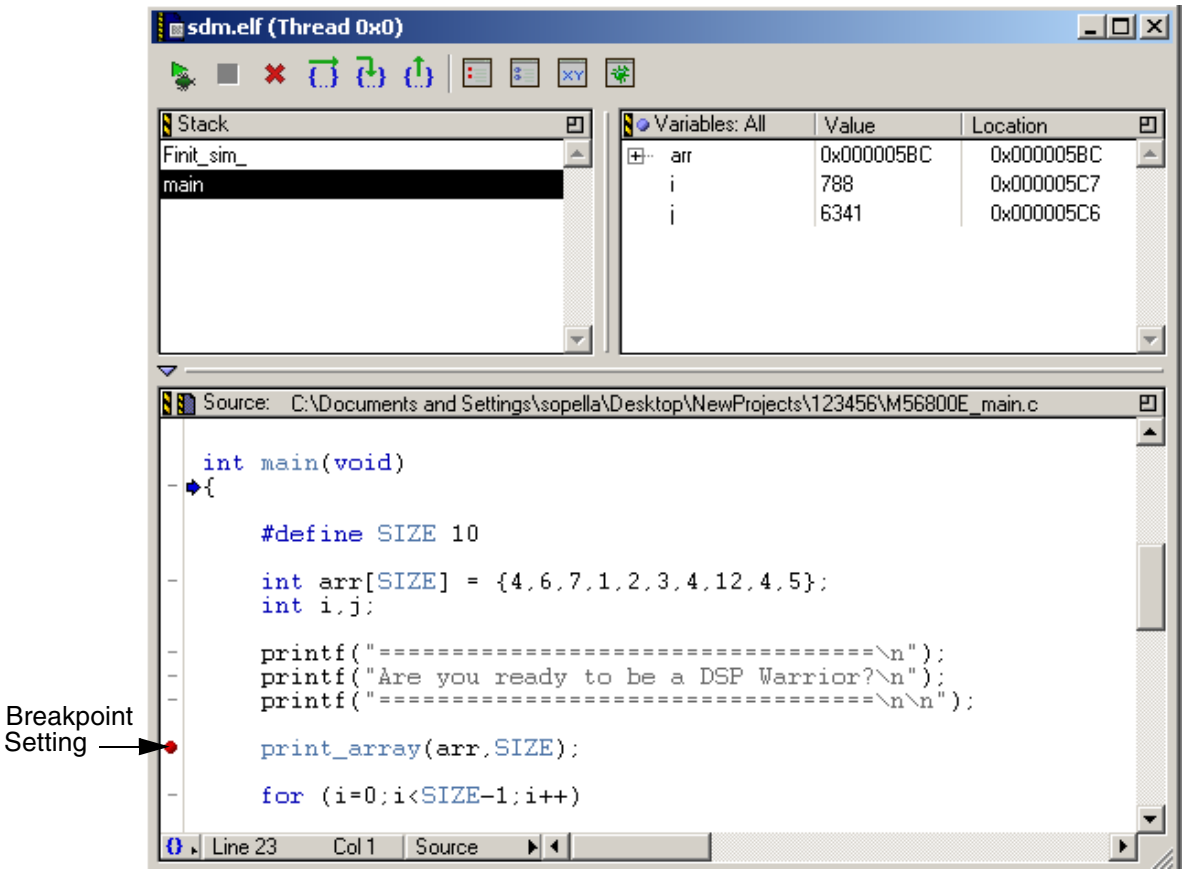
1. **Locate the code line.**

   Scroll through the code in the **Source** pane of the **Program** window until you come across the `main()` function.

2. **Select the code line.**

   Click the gray dash in the far left-hand column of the window, next to the first line of code in the `main()` function. A red dot appears (Figure 7.18), confirming you have set your breakpoint.

Figure 7.18 Breakpoint in the Program Window



NOTE To remove the breakpoint, click the red dot. The red dot disappears.

Setting Watchpoints

For details on how to set and use watchpoints, see the CodeWarrior IDE User's Guide.

NOTE For the DSP56800E only one watchpoint is available. This watchpoint is only available on hardware targets.

Viewing and Editing Register Values

Registers are platform-specific. Different chip architectures have different registers.

1. **Access the Registers window.**

   From the menu bar of the Metrowerks CodeWarrior window, select **View > Registers**.

   Expand the **General Purpose Registers** tree control to view the registers as in Figure 7.19, or double-click on **General Purpose Registers** to view the registers as in Figure 7.20.

Figure 7.19 General Purpose Registers for DSP56800E

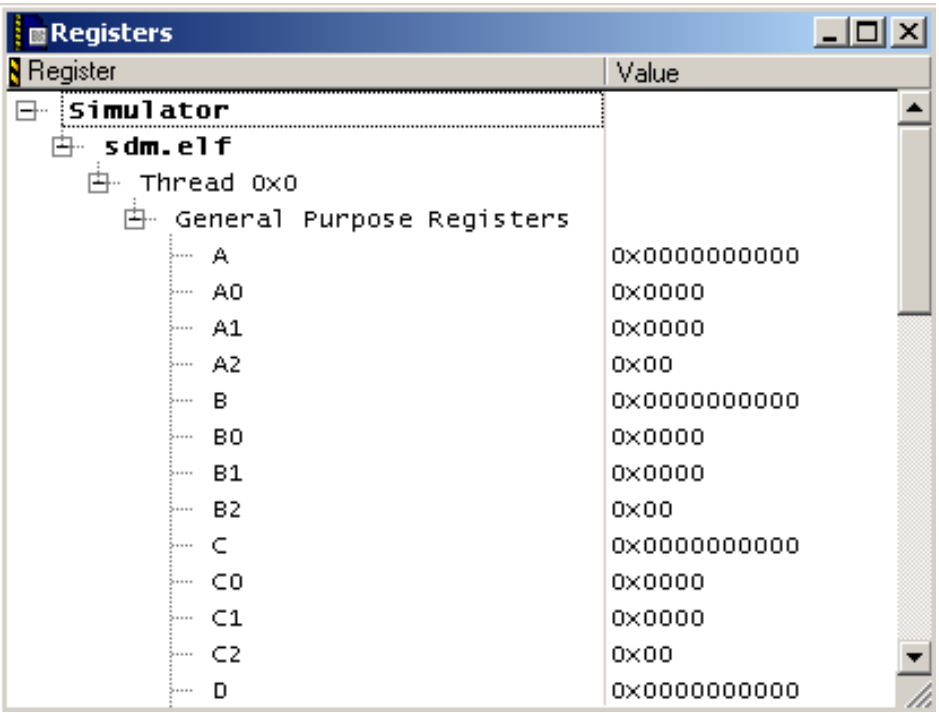
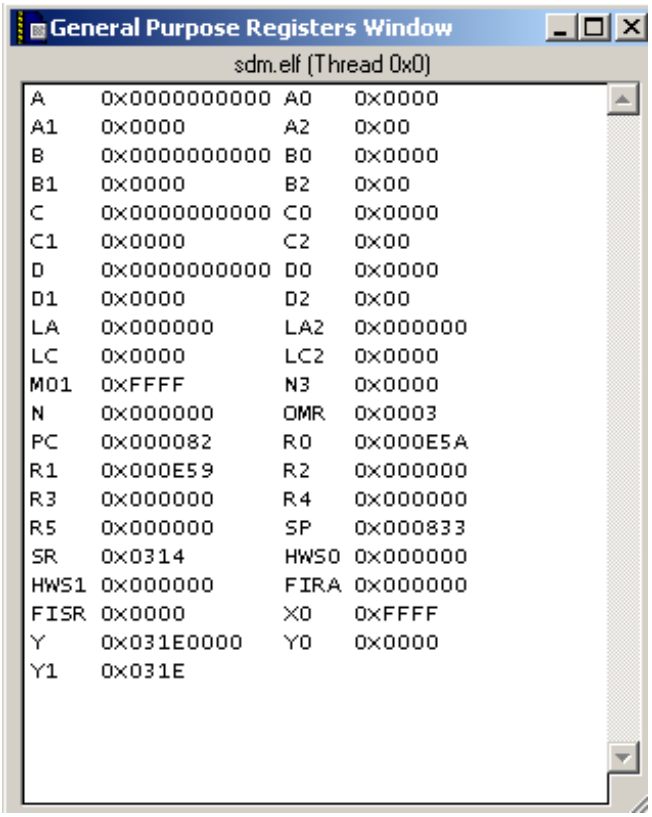


Figure 7.20 General Purpose Registers Window



2. **Edit register values.**

   To edit values in the register window, double-click a register value. Change the value as you wish.

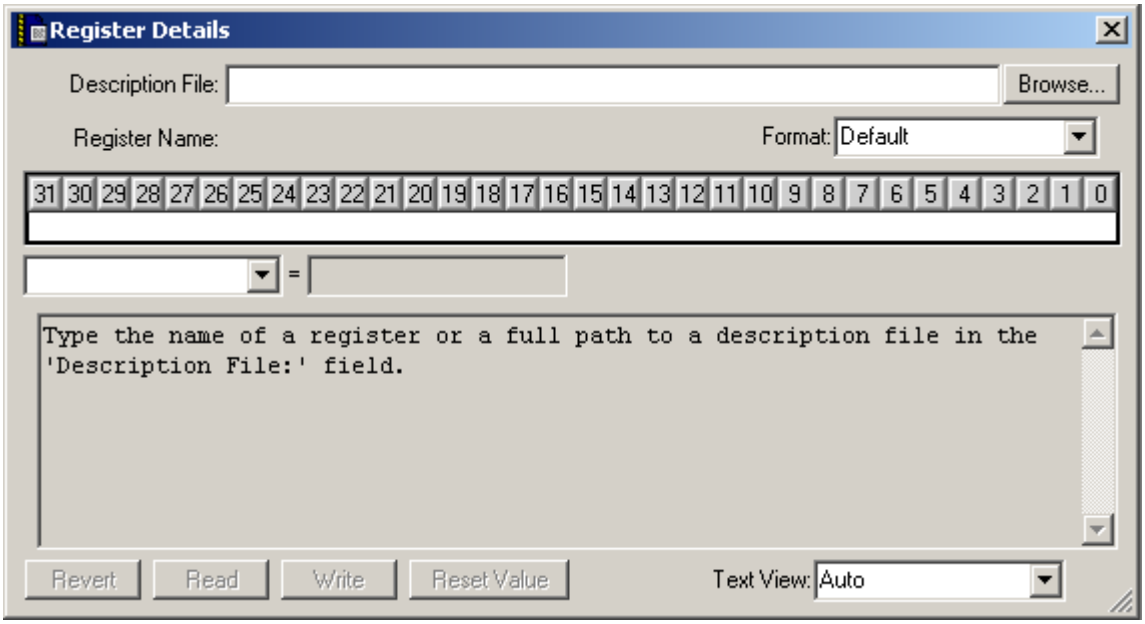
3. **Exit the window.**

   The modified register values are saved.

Register Details Window

From the menu bar of the Metrowerks CodeWarrior window, select **View > Register Details** or in the Registers window (Figure 7.19) double-click on the register. The **Register Details** window appears (Figure 7.21).

Figure 7.21 Register Details Window


Register Details
Description File:
Browse...
Register Name:
Format: Default
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
=
Type the name of a register or a full path to a description file in the 'Description File:' field.
Revert
Read
Write
Reset Value
Text View: Auto

In the **Register Details** window, type the name of the register (e.g., OMR, SR, IPR, etc.) in the **Description File** field. The applicable register and its values appears.

By default, the CodeWarrior IDE looks in the following path when searching for register description files.

```
\CodeWarrior\bin\Plugins\support\Registers\dsp568e\
```

Register description files must end with the `.xml` extension. Alternatively, you can use the **Browse** button to locate the register description files.

Using the **Format** list box in the **Register Details** window, you can change the format in which the CodeWarrior IDE displays the registers.

Using the **Text View** list box in the **Register Details** window, you can change the text information the CodeWarrior IDE displays.

Viewing X: Memory

You can view X memory space values as hexadecimal values with ASCII equivalents. You can edit these values at debug time.

On targets that have Flash ROM, you cannot edit those values in the memory window that reside in Flash memory.

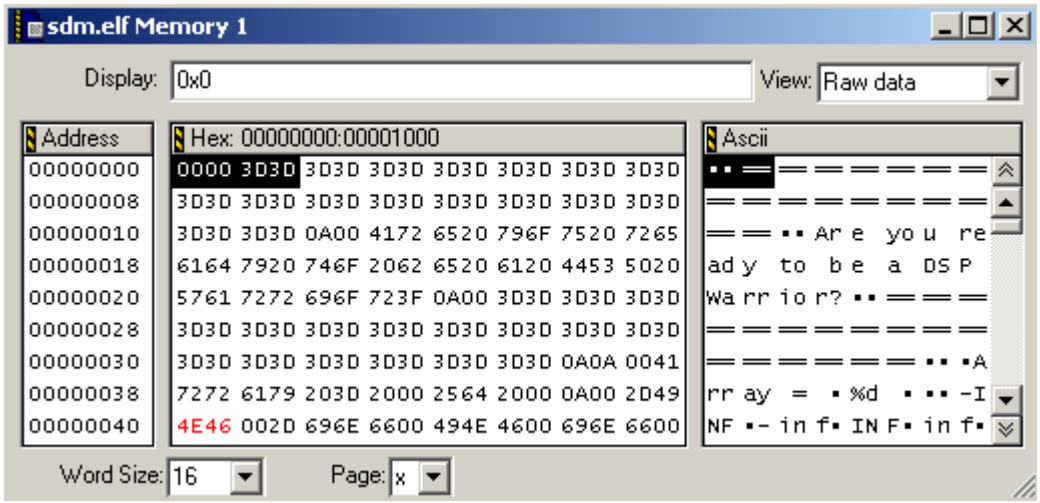
1. **Locate a particular address in program memory.**

   From the menu bar of the Metrowerks CodeWarrior window, select **Data > View Memory**.

NOTE The **Source** pane in the **Program** window needs to be the active one in order for the **Data > View Memory** to be activated.

The **Memory** window appears (Figure 7.22).

Figure 7.22 View X:Memory Window



2. **Select type of memory.**

   Locate the **Page** list box at the bottom of the **View Memory** window. Select **X** for X Memory.

3. **Enter memory address.**

   Type the memory address in the **Display** field located at the top of the **Memory** window.

   To enter a hexadecimal address, use standard C hex notation, for example, 0x0.

NOTE You also can enter the symbolic name whose value you want to view by typing its name in the **Display** field of the **Memory** window.

NOTE The other view options (Disassembly, Source and Mixed) do not apply when viewing X memory.

Viewing P: Memory

You can view P memory space and edit the opcode hexadecimal values at debug time.

NOTE On targets that have Flash ROM, you cannot edit those values in the memory window that reside in Flash memory.

1. **Locate a particular address in program memory.**

   To view program memory, from the menu bar of the Metrowerks CodeWarrior window, select **Data > View Memory**.

   The **Memory** window appears (Figure 7.22).

2. **Select type of memory.**

   Locate the **Page** list box at the bottom of the **View Memory** window. Select **P** for P Memory.

3. **Enter memory address.**

   Type the memory address in the **Display** field located at the top of the **Memory** window.

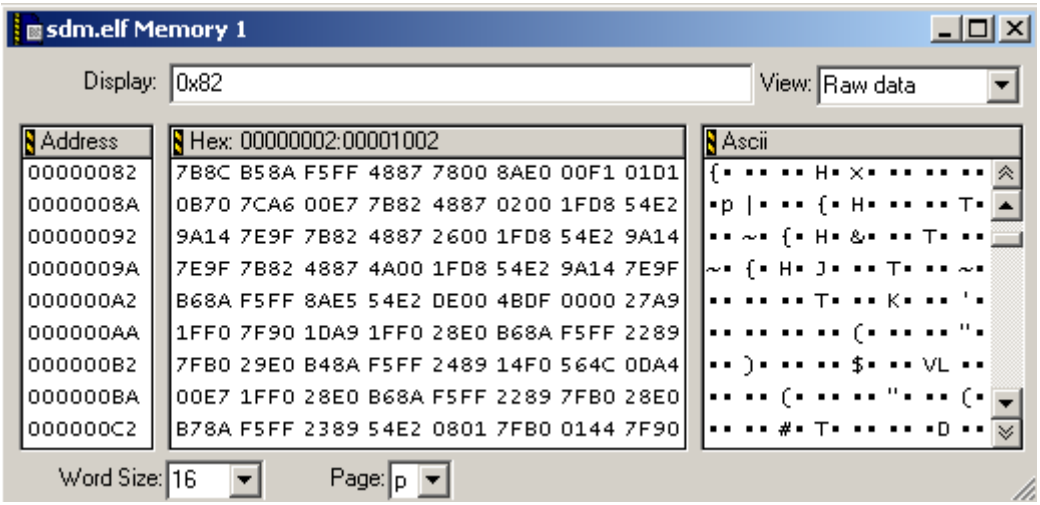
   To enter a hexadecimal address, use standard C hex notation, for example: `0x82`.

4. **Select how you want to view P memory.**

   Using the **View** list box, you have the option to view P Memory in four different ways.

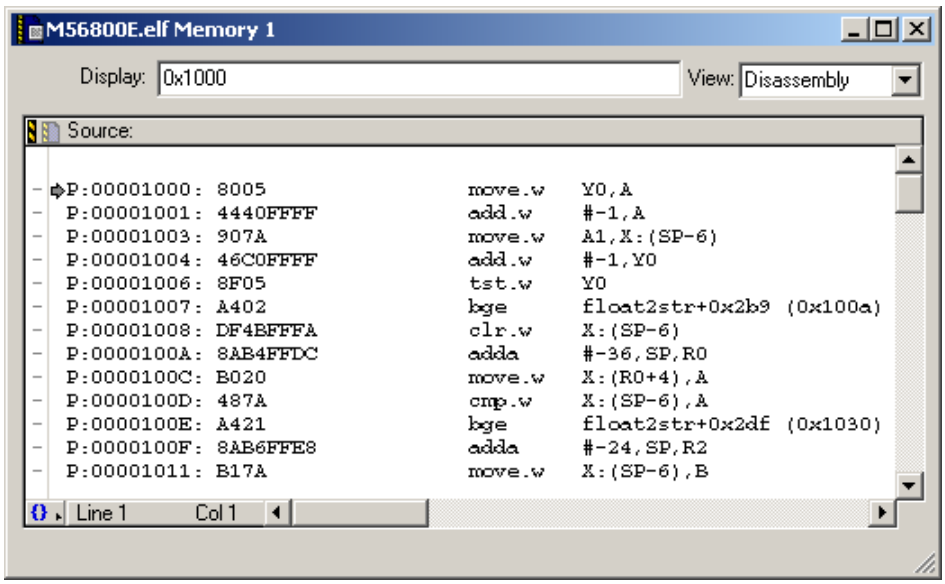
   - **Raw Data** (Figure 7.23).

Figure 7.23 View P:Memory (Raw Data) Window



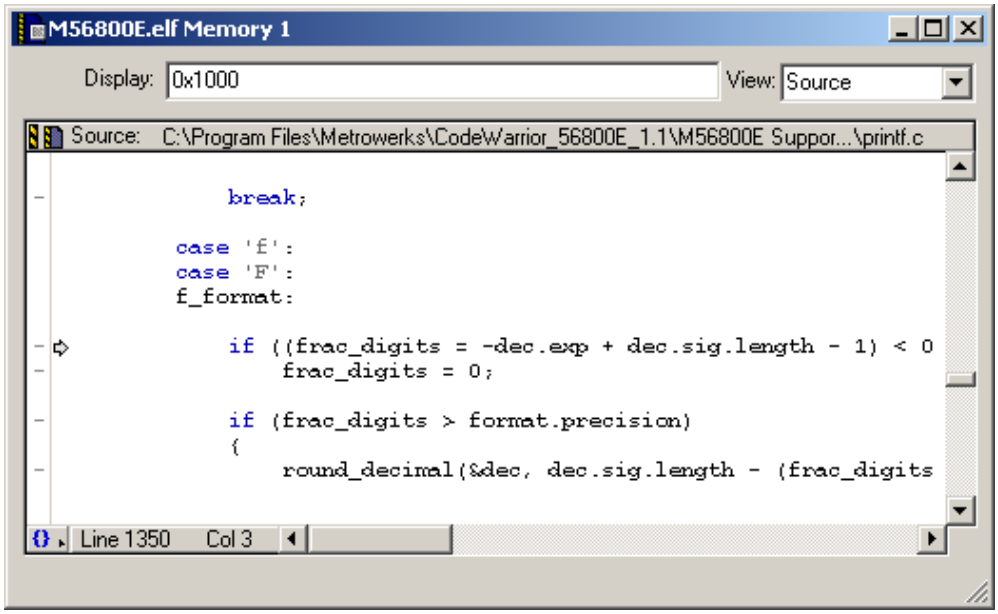
- **Disassembly** (Figure 7.24).

Figure 7.24 View P:Memory (Disassembly) Window



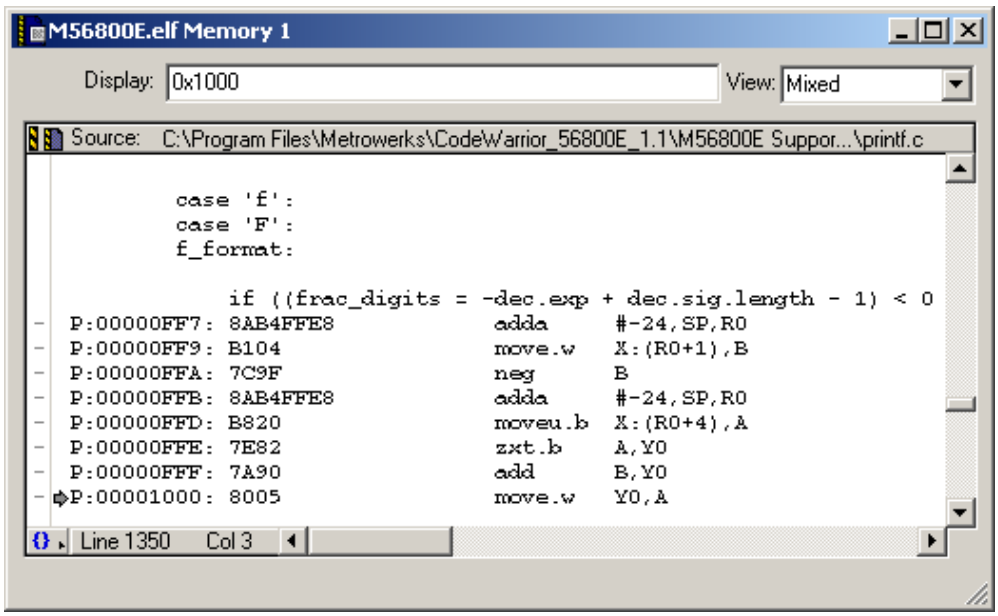
- **Source** (Figure 7.25).

Figure 7.25 View P:Memory (Source) Window



- **Mixed** (Figure 7.26).

Figure 7.26 View P:Memory (Mixed) Window



Loading a .elf File without a Project

You can load and debug a `.elf` file without an associated project. To load a `.elf` file for debugging without an associated project:

1. **Launch the CodeWarrior IDE.**
2. **Choose File > Open and specify the file to load in the standard dialog box that appears.**

   Alternatively, you can drag and drop a `.elf` file onto the IDE.
3. **You may have to add additional access paths in the Access Path preference panel in order to see all of the source code.**
4. **Choose Project > Debug to begin debugging the application.**

NOTE When you debug a `.elf` file without a project, the IDE sets the **Build before running** setting on the Build Settings panel of the IDE Preference panels to Never. Consequently, if you open another project to debug after debugging a `.elf` file, you must change the **Build before running** setting before you can build the project.

The project that the CodeWarrior tools uses to create a new project for the given `.elf` file is `56800E_Default_Project.xml` and is located in the path:

```
CodeWarrior\bin\plugins\support directory
```

You can create your own version of this file to use as a default setting when opening a `.elf` file:

1. **Create a new project with the default setting you want.**
2. **Export the project to xml format.**
3. **Rename the xml format of the project to 56800E_Default_Project.xml and place it in the support directory.**

NOTE Back up or rename the original version of the default xml project before overwriting it with your own customized version.

Command-Line Debugging

In addition to using the regular CodeWarrior IDE debugger windows, you also can debug on the command-line. When you debug on the command-line, you can use:

- Commands included in the Tcl script language
- Additional debugging commands that are specific to the debugger

Tcl Support

This section describes how the command-line debugger handles Tcl support.

Automatically resolving clashing commands

Several command-line debugging commands clash with built-in Tcl commands. The command-line debugger resolves them as shown in Table 7.2 when the mode is set to auto.

Table 7.2 Resolving Clashing Commands

Command	Resolution
load	When you enter the command with one argument containing `.eld` or `.mcp`, the command-line debugger loads the project. Otherwise, the debugger calls the Tcl load command.
break	When you enter the command with no argument and in a script file, the command-line debugger calls the built-in Tcl break command. Otherwise, the debugger uses the break command to control breakpoints.
close	When you enter the close command with no argument, the command-line debugger closes the current debugging session. Otherwise, the debugger calls the built-in Tcl close command.

Tcl support for executing script files

Tcl usually executes a script file as one large block; Tcl returns only after the entire file executes. However, the `run` debugging command executes script files line by line. If a particular line is not a complete Tcl command, the `run` command appends the next line until it gets a complete Tcl script block.

For example, the Tcl `source` command executes the script in Listing 7.1 as one block, but the `run` debugging command executes it as two blocks: the `set` statement and the `while` loop.

Listing 7.1 Example Tcl Script

```
set x 0;

   while {$x < 5}

  {
    puts "x is $x";

    set x [expr $x + 1]

  }
```

NOTE The `run` debugging command synchronizes the debug events between blocks in a script file. For example, after a `go`, `next`, or `step` debugging command, `run` polls the debugging thread state and refrains from executing the next line or block until the debugging thread stops. However, the Tcl `source` command does not consider the state of the debugging thread. Consequently, use the `run` debugging command to execute script files that contain these debugging commands: `debug`, `go`, `next`, `stop`, and `kill`.

Tcl support for using a start-up script

You can use a start-up script with the command-line debugger. (You can specify command-line debugger commands in the script. For example, you might want to set an alias or a color configuration.)

Name the start-up script `tcld.tcl`. The command-line debugger executes the start-up script the first time you open the command-line debugger window, provided that the file is in the correct directory for the host platform you are using. For Windows, place `tcld.tcl` in the Windows installed directory.

NOTE There is no synchronization of debug events in the startup script. Consequently, add the `run` debugging command to the startup script and place the following debugging commands in another script to execute them: `debug`, `go`, `stop`, `kill`, `next`, and `step`.

Command-Line Debugging Tasks

This section describes some tasks for command-line debugging.

Open a Command-Line Debugging Window

To open a command-line debugging window, choose **Debug > Command Line Debugger**.

When the debugging window opens, it displays several command hints at the bottom of the window.

Enter a Single Command-Line Debugging Command

To enter a single debugging command:

1. **Type a command (or its shortcut followed by a space) on the command line.**

   (For example, the shortcut for the `break` command is `b`.)
2. **If needed, type any options, separating them from the command and each other with spaces.**
3. **Press Enter.**

Enter Multiple Command-Line Debugging Commands

To enter multiple debugging commands:

1. **Decide which commands (Tcl and debugger-specific) to use.**
2. **Type the commands into a file.**

3. **Save the file with a `.tcl` extension to indicate that it is a Tcl script.**
4. **Enter the `run` command to run the script.**

View Debugging Command Hints

You can view debugging command hints as follows:

- To view the hint for a particular debugger-specific command, type the command followed by a space.
- The hint shows the syntax for the remainder of the command.
- To scroll through all of the debugger-specific commands that you can use on the command line, press the space bar when the cursor is at the start of the command line in the debugging window.

The highlighted portions of the commands indicate shortcuts that can be used for commands. Press the space bar after typing a shortcut to complete the command automatically.

Repeat a Command-Line Debugging Command

To execute a debugging command again in the command-line debugging window:

1. **Type the debugging command and press Enter.**

   This executes the command the first time.

2. **Press Enter again.**

   This executes the same command again.

   Alternatively, type an exclamation point (!) followed by the ID number of the command and press Enter.

NOTE To see the ID numbers of commands, execute the `history` debugging command.

Review Previously Entered Commands

To sequentially review previously entered commands, press the Up-Arrow and Down-Arrow keys.

Clear a Command from the Command Line

To clear a command from the command line that you have typed but not yet executed, press the Escape key.

Stop an Executing Script

To stop a script that is executing, press the Escape key.

Switch between Insert and Overwrite Mode

To switch between insert and overwrite mode when entering commands on the command line, press the Insert key.

Scroll Text in the Command-Line Debugging Window

The scrolling line number can be set by the `config` debugging command.

To scroll text in the command-line debugging window:

- To scroll up one screen of text, press the Page Up key.
- To scroll down one screen of text, press the Page Down key.

NOTE By default, the number of lines scrolled by the Page Up and Page Down keys is the number of lines displayed in the debugging window. If you resize the window, the number of lines scrolled changes accordingly. You also can use the debugger-specific `config` command to change the number of lines scrolled by the Page Up and Page Down keys.

- To scroll up one line of text, press Ctrl-Up-Arrow key.
- To scroll down one line of text, press Ctrl-Down-Arrow key.
- To scroll left one column, press Ctrl-Left-Arrow key.
- To scroll right one column, press Ctrl-Right-Arrow key.
- To scroll to the beginning of the displayed buffer, press Ctrl-Home.
- To scroll to the end of the displayed buffer, press Ctrl-End.

Copy Text from the Command-Line Debugging Window

To copy text from the window to the clipboard:

1. **Drag your mouse over the text to copy.**
2. **Press Enter or choose Edit > Copy.**

Paste Text into the Command-Line Debugging Window

To paste text from the clipboard into the window:

1. **Place the mouse cursor on the command line.**
2. **Click the right mouse button or choose Edit > Paste.**

Command-Line Debugging Commands

This section describes the command-line debugging commands.

NOTE The default number base for entering commands and displaying registers and memory is hexadecimal. You can change it with the radix command, or you can override it when entering an individual value. To specify a hexadecimal constant, precede the constant with a dollar sign ($). To specify a decimal constant, precede the constant with a grave accent (`). To specify a binary value, precede the constant with a percent sign (%). To specify a fraction value, precede the constant with a caret (^).

alias

Use the `alias` debugging command to:

- Create a pseudonym for a debugging command
- Remove a pseudonym for a debugging command
- List all currently defined aliases

The syntax for the alias command follows:

`al[ias]` [*alias_name*] [*alias_definition*]

Table 7.3 shows examples of the `alias` command.

Table 7.3 Debugging Command Examples: alias

Example	Description
`alias .. cd ..`	This example creates a command named `..` to go to the parent directory.
`alias`	This example lists all the currently set aliases.
`alias ..`	This example removes a previously specified alias (named `..`).

break

Use the `break` debugging command to:

- Set a breakpoint
- Remove a breakpoint
- Display all currently set breakpoints

The syntax for the `break` command follows:

```
b[reak] [func_name | machine_addr] |
        [file_name line_num [column_number]] |
        [func_name | brkpt_num off]
```

Table 7.4 shows examples of the `break` command.

Table 7.4 Debugging Command Examples: break

Example	Description
`break foo`	This example sets a breakpoint on the function `foo`.
`break foo off`	This example removes the breakpoint from the function `foo`.
`break p:$1048a`	This example sets a breakpoint on the machine address `1048a`.
`break`	This example displays all the breakpoints.

Table 7.4 Debugging Command Examples: break (*continued*)

Example	Description
`break #4 off`	This example removes breakpoint number 4. (To determine the number assigned to a particular breakpoint, execute the `break` command.)
`break main.c `15`	This example sets a breakpoint on line 15 in main.c (Please note:The filename argument is case-sensitive. You may get an error message if it is typed incorrectly.)

bringtofront

Use the `bringtofront` debugging command to indicate whether to always display the command-line debugging window in front of all other windows on the screen.

The syntax for the `bringtofront` command follows:

```
bri[ngtofront] [on |off]
```

Table 7.5 shows examples of the `bringtofront` command.

Table 7.5 Debugging Command Examples: bringtofront

Example	Description
`bringtofront`	This example toggles the current `bringtofront` setting of the window.
`bringtofront on`	This example sets the command-line debugger window to always display in front of other windows.

cd

Use the `cd` debugging command to change to a different directory or display the current directory.

When typing a directory name, you can press the Tab key to complete the name automatically.

You can use an asterisk as a wild card when entering directory names.

The syntax for the `cd` command follows:

`cd [path]`

Table 7.6 shows examples of the `cd` command.

Table 7.6 Debugging Command Examples: cd

Example	Description
`cd`	This example displays the current directory.
`cd c:/`	This example changes the directory to the root directory of the `C:` drive.
`cd d:/mw/0622/test`	This example changes the directory to the specified directory on the `D:` drive.
`cd c:/p*s`	This example uses a wild card character (*) to change the current directory to a different directory on the specified drive. For example, if there is a directory named `Program_Files` in the root directory of the `C:` drive, this example changes the current directory to that directory.

change

Use the `change` debugging command to change the contents of registers or memory locations.

You can change the contents of:

- A single register
- A block of registers
- A single memory address
- A block of memory addresses

The syntax for the `change` command follows:

`c[hange]` [*register* | *reg_block*
| *address* | *addr_block*]
[*value*]
[8bit | 16bit | 32bit | 64bit]

reg_block `::=` *register_first*`..`*register_last*

addr_block `::=` *address_first*`..`*address_last* |
address`#`*count*

count `::=` a value indicating the number of memory locations whose contents to change

Table 7.7 shows examples of the `change` command.

Table 7.7 Debugging Command Examples: change

Example	Description
`change R1 $123`	This example changes the contents of `R1` to `123`.
`change R1..R5 $5432`	This example changes the contents of `R1` through `R5` to `5432`.

Table 7.7 Debugging Command Examples: change (*continued*)

Example	Description
`change p:10..17 3456`	This example changes p memory address `10` through `17` to `3456`.
`change p:18..1f $03456`	This example changes p memory addresses `18` through `1f` to `00003456`.

When you change the contents of one or more memory locations, you do not have to specify the memory access mode (whether the mode is eight-bit, 16-bit, 32-bit, or 64-bit).

If you do not specify the memory access mode, the debugger determines it as follows:

- If *`value`* is a fractional value, the mode is 16-bit.
- If *`value`* is a hexadecimal value, the debugger determines the mode as shown in Table 7.8:

Table 7.8 Memory Access Mode for Hexadecimal Values

Memory Access Mode	When value Length Is...	Examples
Eight-bit (8bit)	`length <= 2`	`$1,$12,$01`
16-bit (16bit)	`2 < length <= 4`	`$0001,$123`
32-bit (32bit)	`4 < length <= 8`	`$00000123,$1234567`
64-bit (64bit)	`length > 8`	`$123456789`

- If *`value`* is a decimal value, the debugger determines the mode as shown in Table 7.9:

Table 7.9 Memory Access Mode for Decimal Values

Memory Access Mode	When value Is...	Examples
Eight-bit (8bit)	*value* `<= 0xff`	`0,54,255`
16-bit (16bit)	*value* `> 0xff`	`256,65535,1000`
32-bit (32bit)	`0xffff <` *value* `<= 0xffffffff`	`65536,3253532`
64-bit (64bit)	*value* `> 0xffffffff`	`4294967296`

cls

Use the `cls` debugging command to clear the command-line debugging window.

The syntax for the `cls` command follows:

```
cl[s]
```

close

Use the `close` debugging command to close the opened default project.

The syntax for the `close` command follows:

```
clo[se]
```

config

Use the `config` debugging command to configure the command-line debugging window. You can configure these items:

- Window colors
- Scrolling size
- Mode
- The default build target
- The hexadecimal prefix
- The memory identifier
- The processor name
- The subprocessor name

In addition, you can perform these actions:

- Get the default build target name
- Get the default project name

The syntax for the `config` command follows:

```
conf[ig]
  [ c[olor] [r | m | c | s | e | n]
          text_color [background_color] |
```

```
m[ode] [ dsp | tcl | auto] |
s[croll] number_of_lines |
h[exprefix] hexadecimal_prefix |
mem[identifier] memory_identifier |
p[rocessor] processor_name [subprocessor_name] ]
```

text_color ::= [*R_value* *G_value* *B_value*]

background_color ::= [*R_value* *G_value* *B_value*]

R_value ::= the R value of an RGB color

G_value ::= the G value of an RGB color

B_value ::= the B value of an RGB color

NOTE The valid values to specify an RGB color are from 0 through 255.

number_of_lines ::= the number of lines to scroll

Table 7.10 shows examples of the `config` command.

Table 7.10 Debugging Command Examples: config

Example	Description
`config`	Display all current configuration status information.
`config c e $ff $0 $0`	Set the error text color to red.
`config c r $0 $0 $0 $ff $ff $ff`	Set the register display color to black and the background color to white.
`config s $10`	Set the page scrolling size to decimal 16 lines.
`config m dsp`	Set the command-line debugging window to dsp mode; use the command-line debugging commands when clashing.
`config m tcl`	Set the command-line debugging window to Tcl mode; use the Tcl commands when clashing.

Table 7.10 Debugging Command Examples: config (*continued*)

Example	Description
`config m auto`	Set the command-line debugging window to auto mode; resolve clashing automatically.
`config hexprefix 0x`	Show hexadecimal numbers with `0x` prefix.
`config memidentifier m`	Set the memidentifier to m.
`config processor Dsp568E-Generic`	Set the processor to DSP56800E
`config target`	Get the default build target name.
`config project`	Get the default project name.
`config target sdm`	Change the default build target to `sdm`.

copy

Use the `copy` debugging command to copy the contents of a memory address or block of addresses to another memory location.

The syntax for the `copy` command follows:

`co[py]` *addr_group addr*

addr_group `::=` *`address`* | *addr_block*

addr_block `::=` *address_first..address_last* |
address#count

count ::= a value indicating the number of memory locations

The *addr_group* symbol is the location (or locations) from which the command copies the contents. The *`addr`* variable specifies the first address in memory to which the command copies the contents.

Table 7.11 shows examples of the `copy` command.

Table 7.11 Debugging Command Examples: copy

Example	Description
`copy p:00..1f p:30`	This example copies the contents of p memory addresses `00` through `1f` to a contiguous block of p memory beginning at address `30`.
`copy p:20#10 p:50`	This example copies the contents of 10 consecutive p memory locations that start at address `20` to a contiguous block of p memory beginning at address `50`.

debug

Use the `debug` command to start a command-line debugging session for a project.

The syntax for the `debug` command follows:

`de[bug] [`*project_file_name*`]`

Table 7.12 shows examples of the `debug` command.

Table 7.12 Debugging Command Examples: debug

Example	Description
`debug`	This example starts a debugging session for the open default project.
`debug des.mcp`	This example starts a debugging session for the project named `des.mcp`.

dir or ls

Use the `dir` debugging command to list the contents of a directory when developing on a Windows host. Use the same syntax that you use for the operating system `dir` command.

NOTE You can use the `dir` debugging command the same way you use the `dir` OS command with one exception. You cannot use any option that requires user input from the keyboard (such as `/p` for the `dir` OS command).

Table 7.13 shows examples of the `dir` and `ls` commands.

Table 7.13 Debugging Command Examples: dir

Example	Description
`dir`	This example lists all files in the current directory.
`dir *.txt`	This example lists all files in the current directory that have a file extension of `.txt`.
`dir c:/tmp`	This example lists all files in the `tmp` directory of the `C:` drive.
`dir /ad`	This example lists only the subdirectories in the current directory.
`ls /usr`	This example lists the contents of the subdirectory called `usr`.

disassemble

Use the `disassemble` debugging command to disassemble the instructions in the specified memory block.

The syntax for the `disassemble` command follows:

`di[sassemble]` *addr_block*

addr_block `::=` *address_first*`..`*address_last* | *address*`#`*count*

count ::= a value indicating the number of memory locations

Table 7.14 shows examples of the `disassemble` command.

Table 7.14 Debugging Command Examples: disassemble

Example	Description
`disassemble`	Disassembles instructions from PC (if changed)/last address.
`disassemble p:0..20`	Disassembles program memory address block 0 to 20.
`disassemble p:$50#10`	Disassembles 10 memory locations starting at memory map hexadecimal 50.

display

Use the `display` debugging command to:

- Display the contents of a register or memory location
- List all the register sets on a target
- Add one or more register sets, registers, or memory locations to the default display items
- Remove one or more register sets, registers, or memory locations from the default display items

The memory output radix is specified by the `radix` command.

When you display registers or memory locations, the `display` command returns the values to Tcl. Consequently, you can embed the display command to Tcl as follows:

```
set r0 [display r0]           ; puts $r0 ;
set r0M [display p:$r0 32bit] ; puts $r0M
set r0r1 [display r0..r1]     ; puts $r0r1 ;
```

By default, the `display` command displays memory as 16-bits per unit; you can change that by specifying unit size as `8bit`, `16bit`, `32bit`, `64bit`.

The syntax for the `display` command follows:

```
d[isplay] [ regset ] |
 [on all] |
```

```
[off all] |
[off id_number ] |
[on reg_group | reg_block | addr_group [8bit | 16bit | 32bit | 64bit]] |
[off reg_group | reg_block | addr_group [8bit | 16bit | 32bit | 64bit]]
```

reg_group ::= a list of register sets separated by spaces

reg_block ::= *register_first*..*register_last*

addr_group ::= address | *addr_block*

addr_block ::= *address_first*..*address_last* |
address#count

count ::= a value indicating the number of memory locations

Table 7.15 shows examples of the display command.

Table 7.15 Debugging Command Examples: display

Example	Description
display	Displays the default items (for example, register sets). The command-line debugger executes the display command whenever program execution stops.
display on	Lists the default display items.
display regset	Lists all the available register sets on the target chip.
display on <regset1><regset2>...	Add regset1 and regset2 register sets to the default display items.
display off SIM	Remove the SIM register set from the default display items.
display on ALL	Add all supported register sets to the default display items.
display on p:230#10	Add the specified memory locations to the default display items.

Table 7.15 Debugging Command Examples: display (*continued*)

Example	Description
`display off p:230#10`	Remove the specified memory locations from the default display items.
`display off all`	Remove all the items from the default display items.
`display off #2`	Remove the item whose ID is 2 from the default display items.
`display PC`	Lists the value of register PC and returns to Tcl.
`display R1..R5`	Lists the contents of registers R1 through R5.
`display p:00..$100`	Displays the p memory contents from address 0 to hexadecimal 100.
`display p:00#$200 8bit`	Display hexadecimal 200 memory units' contents from address 0. Access memory in 8bit mode.

evaluate

Use the `evaluate` debugging command to display a C variable type or value.

The syntax of the `evaluate` command follows:

`e[valuate] [ b | d |f | h | u]` *variable*

The following list defines the options for the first parameter, `[ b | d | f | h | u ]`, as follows:

- `b` = binary
- `d` = decimal
- `f` = fraction
- `h` = hex
- `u` = unsigned

The preceding parameter defines the format in which to display the value of the variable.

Table 7.16 Debugging Command Examples: evaluate

Example	Description
`evaluate`	Lists the types for all the variables in the current and global stack.
`evaluate i`	Returns the value of the variable `i`.

exit

Use the `exit` debugging command to close the command-line debugging window.

The syntax for the `exit` command follows:

```
[ex]it
```

go

Use the `go` debugging command to start the program that you are debugging from the current instruction.

The syntax for the `go` command follows:

`g[o] [ all | `*time_period*` ]`

If you execute the `go` command interactively, the command returns immediately. The target program starts to run.

Then you can either wait for the target program to stop executing (for example, on a breakpoint) or type the `stop` debugging command to stop the execution of the target program.

If you execute the `go` command in a script, the command-line debugger polls until the debugger stops (for example, on a breakpoint) and then executes the next command in the script. (If the command-line debugger continues polling indefinitely because the debugging process does not stop, you can stop the script by pressing the Escape key.)

Table 7.17 shows examples of the `go` command.

Table 7.17 Debugging Command Examples: go

Example	Description
`go`	This command returns immediately. The program stops at the first occurrence of a breakpoint. You also can use the `stop` debugging command to break the program.
`go 1`	This command stops polling the target when no breakpoint is reached within 1 second. It also sets a Tcl variable called `$still_running` to 1.
`go all`	This command starts all the target programs when debugging multiple cores.

help

Use the `help` debugging command to display help for the debugging commands in the command-line debugger window.

The syntax for the `help` command follows:

`h[elp] [`*command* | *command_shortcut*`]`

Table 7.18 shows examples of the `help` command.

Table 7.18 Debugging Command Examples:help

Example	Description
`help`	This example lists all the debugging commands.
`help b`	This example displays help for the `break` debugging command.

history

Use the `history` debugging command to list the history of the commands entered during the current debugging session.

The syntax for the `history` command follows:

```
hi[story]
```

hsst_attach_listener

Use the `hsst_attach_listener` command to set up a Tcl procedure that the debugger notifies whenever there is data in a communication channel.

The syntax for `hsst_attach_listener` command follows:

`hsst_a[ttach_listener]` *channel_id tcl_proc_name*

The following example uses the `hsst_attach_listener` command to execute the procedure `call_back` automatically when a communication channel has data available from the target.

```
proc call_back { } {
 global hsst_descriptor;
 global hsst_nmemb;
 global hsst_size;
 puts [ hsst_read $hsst_size $hsst_nmemb
$hsst_descriptor ]
}
set cid [ hsst_open channel1 ]
hsst_attach_listener $cid call_back;
```

hsst_block_mode

Use the `hsst_block_mode` command to configure a communication channel in blocking mode. Doing so causes all calls to `hsst_read` to block until the requested amount of data is available from the target.

The default setting is for all channels to be in blocking mode.

The syntax for `hsst_block_mode` follows:

`hsst_b[lock_mode]` *channel_id*

The following example configures a channel in blocking mode:

```
hsst_block_mode $cid
```

hsst_close

Use the `hsst_close` debugging command to close a communication channel with the host machine.

The syntax for the `hsst_close` command follows:

`hsst_c[lose]` *channel_id*

The following example closes a channel and sets the result to the variable `$cid`:

```
hsst_close $cid
```

hsst_detach_listener

Use the `hsst_detach_listener` command to detach a listener that had been previously attached for automatic data notification.

The syntax for `hsst_detach_listener` follows:

`hsst_d[etach_listener]` *channel_id*

The following example detaches a listener that previously was attached:

```
hsst_detach_listener $cid
```

hsst_log

Use the `hsst_log` debugging command to log the data to a directory.

The syntax for the `hsst_log` command follows:

`hsst_l[og] cid [` *directory_name* `]`

Table 7.19 shows examples of the `hsst_log` command:

Table 7.19 Debugging Command Examples: hsst_log

Example	Description
`hsst_log cid c:\logdata`	The debugger logs the data to the specified directory.
`hsst_log cid 0`	The debugger turns off the log.

hsst_noblock_mode

Use the `hsst_noblock_mode` command to configure a communication channel in non-blocking mode. Dong so causes all calls to `hsst_read` to return immediately with any available data (limited by the requested size).

The syntax for `hsst_noblock_mode` follows:

`hsst_n[oblock_mode]` *channel_id*

The following example configures a channel in non-blocking mode:

```
set cid [ hsst_open channel1 ]
hsst_noblock_mode $cid
```

hsst_open

Use the `hsst_open` debugging command to open a communication channel with the host machine.

The syntax for the `hsst_open` command follows:

`hsst_o[pen]` *channel_name*

The following example opens a channel and sets the returned ID to the variable `$cid`:

```
set cid [hsst_open ochannel1]
```

hsst_read

Use the `hsst_read` debugging command to read data from an opened communication channel.

The syntax for the `hsst_read` command follows:

`hsst_r[ead]` *size nmemb cid*

The following example uses the `hsst_read` command to read 15 data items (each 1 byte in length) from the channel identified by the variable `$cid`:

```
puts [hsst_read 1 15 $cid]
```

The debugger returns and displays the data.

hsst_write

Use the `hsst_write` debugging command to write data to an opened communication channel.

The syntax for the `hsst_write` command follows:

`hsst_w[rite]` *size data cid*

The following example uses the `hsst_write` command to write `0x1234` as 2 bytes of data to the channel identified by the variable `$cid`:

```
hsst_write 2 0x1234 $cid
```

input

Use the `input` debugging command to map a target memory block to a host file. When a target application reads the memory block, the application reads the contents of the specified host file instead.

The syntax for the `input` command follows:

```
i[nput]
[ id_num | address filename
[ -rd | -rf | -rh | -ru ]] |
[ off ]
```

Specify *`address`* when using the simulator to debug. Specify *`id_num`* when using target hardware to debug.

Choose from the following options to indicate the format of the input file:

- Use `-rd` to indicate that the input file is a decimal file.
- Use `-rf` to indicate that the input file is a fractional file.
- Use `-rh` to indicate that the input file is a hexadecimal file.
- Use `-ru` to indicate that the input file is an unsigned decimal file.

Table 7.20 shows examples of the `input` command.

Table 7.20 Debugging Command Examples: input

Example	Description
`input p:$100 in.dat -RD` (This example is valid only for the simulator.)	This example sets up the input feature so that the simulator gets values from the `in.dat` file in decimal format (specified by `-RD`) and places them in a memory block `p:$100` when the target application reads program memory location `$100`.
`input off`	This example closes all input files and stops the input feature. (The command is the same both when debugging with the simulator or with target hardware.)

kill

Use the `kill` debugging command to close one or all current debugging sessions.

The syntax for the `kill` command follows:

```
k[ill] [all]
```

Table 7.21 shows examples of the `kill` command.

Table 7.21 Debugging Command Examples: kill

Example	Description
`kill`	Kills the current debugging session.
`kill all`	Kills all the debug sessions when debugging multiple cores.

load

Use the `load` debugging command to open a project or load records into memory.

The syntax for the `load` command follows:

`l[oad]` *project_file_name* | *eld_file_name*

or

`l[oad] -h | -b` *file_name* `[` *memory_location* `]`

The following list defines the first parameter of the second version of the load command:

- `-h` = hexadecimal file
- `-b` = binary file

Table 7.22 shows examples of the load command.

Table 7.22 Debugging Command Examples: load

Example	Description
`load des.mcp`	Loads a project named `des.mcp`.
`load des.eld`	Creates a default project from the `des.eld` object file and loads the project.

Example	Description
`load -h dat.lod`	Loads the contents of the hexadecimal file `dat.lod` into memory.
`load -b dat.lod p:$20`	Loads the contents of the binary file `dat.lod` into program memory beginning at `$20`.

log

Use the `log` debugging command to log either the commands that you enter during a debugging session or the entire session (all display entries) during a debugging session.

The syntax for the `log` command follows:

`lo[g] [off] [c | s` *file_name*`]`

Table 7.23 shows examples of the `log` command.

Table 7.23 Debugging Command Examples: log

Example	Description
`log`	This example displays a list of currently opened log files.
`log s session.log`	This example logs all display entries to a file named `session.log`.
`log c command.log`	This example logs the commands that you enter during the debugging session to a file named `command.log`
`log off c`	This example ends command logging.
`log off`	This example ends all logging in the command-line debugging window.

next

Use the `next` debugging command to step over subroutine calls.

If you execute the `next` command interactively, the command returns immediately. The target program starts to run.

Then you can either wait for the target program to stop executing (for example, on a breakpoint) or type the `stop` debugging command to stop the execution of the target program.

If you execute the `next` command in a script, the command-line debugger polls until the debugger stops (for example, on a breakpoint) and then executes the next command in the script. (If the command-line debugger continues polling indefinitely because the debugging process does not stop, you can stop the script by pressing the Escape key.)

The syntax for the `next` command follows:

```
n[ext]
```

output

Use the output debugging command to map a target memory block to a host file. When the target application writes to the memory block, the application writes the contents to the specified file instead.

The syntax for the `output` command follows:

o[utput]
```
[ address filename
[ -rd | -rf | -rh | -ru ] [-a/-o] ] |
[ off ]
```

Specify *`address`* when using the simulator to debug.

Choose from the following options to indicate the format of the output file:

- Use `-rd` to indicate that the output file is a decimal file.
- Use `-rf` to indicate that the output file is a fractional file.

- Use `-rh` to indicate that the output file is a hexadecimal file.
- Use `-ru` to indicate that the output file is an unsigned decimal file.

Choose from the following options to indicate how to write to the output file:

- Use `-a` to cause the debugger to append to the output file if it already exists.
- Use `-o` to cause the debugger to overwrite the output file if it already exists.

Table 7.24 shows examples of the output command.

Table 7.24 Debugging Command Examples: Output

Example	Description
`output p:$0 out.dat -RD -A` (This example is valid only for the simulator.)	This example stores values (which are written to the memory location `p:0` by the target application) to the file out.dat in decimal format (indicated by `-RD`). The `-A` option appends the values to file `out.dat` if the file already exists.
`output off`	This example closes all output files and stops the output feature.

pwd

Use the `pwd` debugging command to display the working directory.

The syntax for the `pwd` command follows:

```
pwd
```

radix

Use the `radix` debugging command to:

- Display the current default input radix (number base)
- Change the default number base for command entry or for display of registers and memory locations.

Changing the default input radix allows you to enter constants in the chosen radix without typing a radix specifier before each constant.

By default, the command-line debugger uses hexadecimal as the input radix and display radix unless you change them.

NOTE You can override the default input radix when entering an individual value. To specify a hexadecimal constant, precede the constant with a dollar sign ($). To specify a decimal constant, precede the constant with a grave accent (`). To specify a binary value, precede the constant with a percent sign (%). To specify a fraction value, precede the constant with a caret (^).

The syntax for the `radix` command follows:

```
r[adix] [b | d | f | h | u]
          [ register | reg_block | addr_group ]...
```

The following list defines the first parameter, `[ b | d | f | h | u ]`, as follows:

- `b` = binary
- `d` = decimal
- `f` = fraction
- `h` = hex
- `u` = unsigned

The preceding parameter, when not followed by register names or memory locations, specifies the radix to use as the default input radix. When followed by register names or memory locations, the radix is the default display radix when displaying the values contained in the specified register or memory location.

reg_block `::=` *register_first..register_last*

addr_group `::=` *address* | *addr_block*

addr_block `::=` *address_first..address_last* |
address#count

count `::=` a value indicating the number of memory locations

If you enter the `radix` command without any parameters, the debugging window displays the current default input radix.

Table 7.25 shows examples of the `radix` command.

Table 7.25 Debugging Command Examples: radix

Example	Description
`radix`	Displays the currently enabled radix.
`radix D`	Changes the input radix to decimal.
`radix H`	Changes the input radix to hexadecimal.
`radix f r0..r7`	Changes the display radix for the specified registers to fraction.
`radix d x:0#10 r1`	Changes the display radix for the specified register and memory blocks to decimal.

restart

Use the `restart` debugging command to restart the debugging session.

The syntax for the `restart` command follows:

```
[re]start
```

run

Use the `run` debugging command to execute a Tcl script.

This command executes a script file block by block.

NOTE You can use the `run` command to run a script that includes these commands: `load`, `close`, `debug`, `kill`, and `run`. However, the preceding commands cannot reside in a block (such as a loop).

For example, this script is invalid:

```
set x 0
while {$x < 5}
{
load a.mcp
debug
kill
}
```

The syntax for the `run` command follows:

`ru[n]` *file_name*

The following example executes a file named `test.tcl`:

```
run test.tcl
```

save

Use the `save` debugging command to save the contents of specified memory locations to a binary file or a text file in hexadecimal format.

The syntax for the `save` command follows:

`sa[ve] -h | -b` *addr_block* `... filename [-a | -c | -o]`

addr_block `::=` *address_first*`..`*address_last* `|`
address#count

count ::= a value indicating the number of memory locations

The following list defines the first parameter to the `save` command:

- `-h` = write a text file in hexadecimal format
- When you save to a file in hexadecimal text format, the debugger saves the memory location information in the file. Consequently, when you load a file saved in this format, you do not have to specify the memory address.
- `-b` = write a binary file
- When you save to a binary file, the debugger does not save the memory location information in the file. Consequently, when

you load a file saved in this format, you must specify the memory address.

The following list defines the last parameter to the `save` command:

- `-a` = append to an existing file
- `-c` = write if the file does not yet exist
- If the file to which you are trying to save already exists, the `save` command does not overwrite the file. Instead, the `save` command cancels and returns without changing the file.
- `-o` = overwrite an existing file

You can use the Tcl `set` command to assign a name to a particular block of memory. You can then substitute that name instead of typing the specification for the memory block repeatedly.

Table 7.26 shows examples of the `save` command.

Table 7.26 Debugging Command Examples: save

Example	Description
`set addressBlock1 "p:10..`31"` `set addressBlock2 "p:10000#20"` `save -h $addressBlock1 $addressBlock2 hexfile -a`	Dumps the contents of two memory blocks to a text file called `hexfile.lod` in append mode.
`set addressBlock1 "p:10..`31"` `set addressBlock2 "p:10000#20"` `save -b $addressBlock1 $addressBlock2 binfile -o`	Dumps the contents of two memory blocks to a binary text file called `binfile.lod` in overwrite mode.

step

Use the `step` debugging command to step through a program.

The debugger automatically executes the `display` debugging command each time that you invoke the `step` command.

The syntax for the `step` command follows:

```
st[ep] [li | in | into | out]
```

Table 7.27 shows examples of the `step` command.

Table 7.27 Debugging Command Examples: step

Example	Description
`step li`	This example steps one line.
`step in`	This example steps one instruction.
`step into`	This example steps into a function.
`step out`	This example steps out of a function.

stop

Use the `stop` debugging command to stop a running program after invoking a `go`, `step`, or `next` debugging command.

The syntax for the `stop` command follows:

```
s[top] [all]
```

Table 7.28 shows examples of the `stop` command.

Table 7.28 Debugging Command Examples: stop

Example	Description
`stop`	Stops the currently running target program.
`stop all`	Stops all currently running target programs when debugging multiple cores.

switchtarget

When you are performing multi-core or multi-chip debugging, use the `switchtarget` debugging command to list the available debugging sessions and to specify to which session you want to send subsequent debugging commands.

The syntax for the `switchtarget` command follows:

`sw[itchtarget] [`*`index`*`]`

Table 7.29 shows examples of the `switchtarget` command.

Table 7.29 Debugging Command Examples: switchtarget

Example	Description
`switchtarget`	This example lists the currently available debugging sessions.
`switchtarget 0`	Choose the debugging session whose session ID is `0` to send subsequent debugging commands to.

system

Use the `system` debugging command to execute a system command.

NOTE The command-line debugger supports executing system commands that require keyboard input. However, the command-line debugger does not support commands that use the full screen display (such as the DOS edit command).

The syntax for the `system` command follows:

`sy[stem]` *system_command*

The following example runs a system command that deletes all files in the current directory with the `.tmp` file extension:

```
system del *.tmp
```

view

Use the `view` debugging command to change the view mode. You can toggle the view mode between assembly mode and register mode.

The syntax for the `view` command follows:

```
v[iew] [a | r]
```

Table 7.30 shows examples of the `view` command.

Table 7.30 Debugging Command Examples: view

Example	Description
`view`	Toggle the view mode.
`view a`	Set the view mode to assembly mode.
`view r`	Set the view mode to register mode.
`view a $100`	Display the assembly that begins at hexadecimal address `100`.

wait

Use the `wait` debugging command to cause the debugger to wait for the specified amount of time.

The syntax for the `wait` command follows:

`w[ait] [`*milliseconds*`]`

Table 7.31 shows examples of the `wait` command:

Table 7.31 Debugging Command Examples: wait

Example	Description
`wait`	The debugger waits until you press the space bar on the keyboard.
`wait 2`	The debugger waits for two milliseconds.

watchpoint

Use the `watchpoint` debugging command to add, remove, or display a watchpoint.

NOTE Due to hardware resource limitations, you can set only one watchpoint at a time.

The syntax for the `watchpoint` command follows:

`wat[chpoint] [`*`variable_name`* `|` *`watchpoint_id`* `off]`

Table 7.31 shows examples of the `watchpoint` command:

Table 7.32 Debugging Command Examples: watchpoint

Example	Description
`watchpoint`	The debugger displays the watchpoint list.
`watchpoint i`	The debugger adds the variable `i` to the watchpoint list.

System-Level Connect

The CodeWarrior IDE DSP56800E debugger lets you connect to a loaded target board and view system registers and memory. A system-level connect does not let you view symbolic information during a connection.

NOTE The following procedure explains how to connect in the context of developing and debugging code on a target board. However, you can select the **Debug > Connect** command anytime you have a project window open, even if you have not yet downloaded a file to your target board.

To perform a system-level connect:

1. **Select the Project window for the program you downloaded.**
2. **From the menu bar, select Debug > Connect.**

   The debugger connects to the board. You can now examine registers and the contents of memory on the board.

Debugging in the Flash Memory

The debugger is capable of programming flash memory. The programming occurs at launch, during download. The flash programming option is turned on and the parameters are set in the initialization file. This file is specified in the **Debugger>M56800E Target** preference panel. A list of flash memory commands is given in the next section.

The stationery provides an example of how to specify a default initialization file, how to write a linker command file for flash memory, and how to copy initialized data from ROM to RAM using provided library functions.

Flash Memory Commands

The following is a list of flash memory commands that can be included in your initialization file.

set_hfmclkd <value>

This command writes the `value` which represents the clock divider for the flash memory to the hfmclkd register.

The value for the `set_hfmclkd` command depends on the frequency of the clock. If you are using a supported EVM, this value should not be changed from the value provided in the default

initialization file. However, if you are using an unsupported board and the clock frequency is different from that of the supported EVM, a new value must be calculated as described in the user's manual of the particular processor that you are using.

NOTE The `set_hfmclkd`, `set_hfm_base`, and at least one `add_hfm_unit` command must exist to enable flash programming. All other flash memory commands are optional.

set_hfm_base <address>

This command sets the address of `hfm_base`, which is where the flash control registers are mapped in X: memory.

NOTE The `set_hfm_base` and `add_hfm_unit` commands should not be changed for a particular processor. Their values will always be the same.

set_hfm_config_base <address>

This command sets the address of `hfm_config_base`, which is where the flash security values are written in program flash memory. If this command is present, the debugger used the address to mimic part of the hardware reset behavior by copying the protection values from the configuration field to the appropriate flash control registers.

add_hfm_unit <startAddr> <endAddr> <bank> <numSectors> <pageSize> <progMem> <boot> <interleaved>

This command adds a flash unit to the list and sets its parameters.

NOTE The `set_hfm_base` and `add_hfm_unit` commands should not be changed for a particular processor. Their values will always be the same.

set_hfm_erase_mode units | pages | all

This command sets the erase mode as `units`, `pages` or `all`. If you set this to `units`, the units that are programmed are mass erased. If set this to `pages`, the pages that are programmed are erased. If you set this to `all`, all units are mass erased including those that have not been programmed. If you omit this command, the erase mode defaults to the unit mode.

set_hfm_verify_erase 1 | 0

If you set this to `1`, the debugger verifies that the flash memory has been erased, and alerts you if the erase failed. If this command is omitted, the flash erase is not verified.

set_hfm_verify_program 1 | 0

If you set this to `1`, the debugger verifies that the flash has been programmed correctly, and alerts you if the programming failed. If you omit this command, flash programming is not verified.

Unlock Flash Security Features

The flash memory security features of the DSP56800E allow the processor to be "locked out" and thus disable access through the EOnCE port. If the debugger attempts to connect to a secure device, and the initialization file is set up for flash programming, the debugger will then disable the security feature by issuing a command that erases all of the flash memory.

Notes for Debugging on Hardware

The following list are some things to be aware of when debugging on a hardware target.

- There is only one hardware breakpoint available, which is shared by IDE breakpoints (when the Breakpoint Mode is set to hardware in the **M56800E Target** panel), watchpoints, and EOnCE triggers. Only one of these may be set at a time.
- When a hardware breakpoint trigger is set to react to an instruction fetch (IDE hardware breakpoint or EOnCE trigger) be aware that the hardware will react to the fetch whether or not

the fetched instruction is executed. For example, if a hardware breakpoint is set just after a loop, the processor will stop with the execution point inside the loop. This is because the target instruction will be fetched while the program is in the loop due to the large pipeline. A branch will occur to facilitate the loop; however, the processor will stop because the target instruction has already been fetched.

- The M56800E cannot single step over certain two and three-word uninterrupted sequences. However, the debugger compensates using software breakpoints and the trace buffer to allow single stepping in these situations. But, if these techniques cannot be used (e.g., debugging in ROM or the trace buffer in use) single stepping over these sequences results in the processor executing each instruction in the sequence before stopping. The execution will be correct. Just be aware of this "slide" in these situations.

8

High-Speed Simultaneous Transfer

High-Speed Simultaneous Transfer (HSST) facilitates data transfer between low-level targets (hardware or simulator) and host-side client applications. The data transfer occurs without stopping the core.

The host-side client must be an IDE plug-in or a script run through the command-line debugger.

When the customer links their application to the target side hsst lib, the debugger detects that the customer wants to use hsst and automatically enables hsst communications.

NOTE To use HSST, you must launch the target side application through the debugger.

Host-Side Client Interface

This section describes the API calls for using High-Speed Simultaneous Transfer (HSST) from your host-side client application.

At the end of this section, an example of a HSST host-side program is given (Listing 8.1 on page 258).

hsst_open

A host-side client application uses this function to open a communication channel with the low-level target. Opening a channel that has already been opened will result in the same channel ID being returned.

```
HRESULT hsst_open (
    const char* channel_name,
    size_t *cid );
```

`channel_name`

This parameter specifies the communication channel name.

`cid`

This parameter specifies the channel ID associated with the communication channel.

Returns This function returns `S_OK` if the call succeeds or `S_FALSE` if the call fails.

hsst_close

A host-side client application uses this function to close a communication channel with the low-level target.

```
HRESULT hsst_close ( size_t channel_id ) ;
```

`channel_id`

This parameter specifies the channel ID of the communication channel to close.

Returns This function returns `S_OK` if the call succeeds or `S_FALSE` if the call fails.

hsst_read

A host-side client application uses this function to read data sent by the target application without stopping the core.

```
HRESULT hsst_read  (
    void *data,
    size_t size,
    size_t nmemb,
    size_t channel_id,
    size_t *read );
```

`data`

This parameter specifies the data buffer into which data is read.

`size`

This parameter specifies the size of the individual data elements to read.

`nmemb`

This parameter specifies the number of data elements to read.

`channel_id`

This parameter specifies the channel ID of the communication channel from which to read.

`read`

This parameter contains the number of data elements read.

Returns This function returns `S_OK` if the call succeeds or `S_FALSE` if the call fails.

hsst_write

A host-side client application uses this function to write data that the target application can read without stopping the core.

```
HRESULT hsst_write (
    void *data,
    size_t size,
    size_t nmemb,
    size_t channel_id,
    size_t *written );
```

`data`

This parameter specifies the data buffer that holds the data to write.

`size`

This parameter specifies the size of the individual data elements to write.

`nmemb`

This parameter specifies the number of data elements to write.

`channel_id`

This parameter specifies the channel ID of the communication channel to write to.

`written`

This parameter contains the number of data elements written.

Returns

This function returns `S_OK` if the call succeeds or `S_FALSE` if the call fails.

hsst_size

A host-side client application uses this function to determine the size of unread data (in bytes) in the communication channel.

```
HRESULT hsst_size  (
    size_t channel_id,
    size_t *unread );
```

`channel_id`

This parameter specifies the channel ID of the applicable communication channel.

`unread`

This parameter contains the size of unread data in the communication channel.

Returns This function returns `S_OK` if the call succeeds or `S_FALSE` if the call fails.

hsst_block_mode

A host-side client application uses this function to set a communication channel in blocking mode. All calls to read from the specified channel block indefinitely until the requested amount of data is available. By default, a channel starts in the blocking mode.

```
HRESULT hsst_block_mode ( size_t channel_id );
```

`channel_id`

This parameter specifies the channel ID of the communication channel to set in blocking mode.

Returns This function returns `S_OK` if the call succeeds or `S_FALSE` if the call fails.

hsst_noblock_mode

A host-side client application uses this function to set a communication channel in non-blocking mode. Calls to read from the specified channel do not block for data availability.

```
HRESULT hsst_noblock_mode ( size_t channel_id );
```

`channel_id`

This parameter specifies the channel ID of the communication channel to set in non-blocking mode.

Returns This function returns `S_OK` if the call succeeds or `S_FALSE` if the call fails.

hsst_attach_listener

Use this function to attach a host-side client application as a listener to a specified communication channel. The client application receives a notification whenever data is available to read from the specified channel.

HSST notifies the client application that data is available to read from the specified channel. The client must implement this function:

```
void NotifiableHSSTClient:: Update (size_t descrip-
    tor, size_t size, size_t nmemb);
```

HSST calls the `Notifiable HSST Client:: Update` function when data is available to read.

```
HRESULT hsst_attach_listener (
    size_t cid,
    NotifiableHSSTClient *subscriber );
```

`cid`

This parameter specifies the channel ID of the communication channel to listen to.

`subscriber`

This parameter specifies the address of the variable of class `Notifiable HSST Client`.

Returns This function returns `S_OK` if the call succeeds or `S_FALSE` if the call fails.

hsst_detach_listener

Use this function to detach a host-side client application that you previously attached as a listener to the specified communication channel.

```
HRESULT hsst_detach_listener ( size_t cid );
```

`cid`

This parameter specifies the channel ID of the communication channel from which to detach a previously specified listener.

Returns This function returns `S_OK` if the call succeeds or `S_FALSE` if the call fails.

hsst_set_log_dir

A host-side client application uses this function to set a log directory for the specified communication channel.

This function allows the host-side client application to use data logged from a previous High-Speed Simultaneous Transfer (HSST) session rather than reading directly from the board.

After the initial call to `hsst_set_log_dir`, the CodeWarrior software examines the specified directory for logged data associated with the relevant channel instead of communicating with the board to get the data. After all the data has been read from the file, all future reads are read from the board.

To stop reading logged data, the host-side client application calls `hsst_set_log_dir` with `NULL` as its argument. This call only affects host-side reading.

```
HRESULT hsst_set_log_dir (
    size_t cid,
    const char* log_directory );
```

`cid`

This parameter specifies the channel ID of the communication channel from which to log data.

`log_directory`

This parameter specifies the path to the directory in which to store temporary log files.

Returns This function returns `S_OK` if the call succeeds or `S_FALSE` if the call fails.

HSST Host Program Example

In Listing 8.1 the host is the IDE plugin (DLL) to the interface with the HSST target (DSP56800E) project. This establishes data transfer between the host (your computer) and the target (the DSP56800E board).

NOTE Before launching the program, the IDE plugin needs to be created and placed in the folder: `CodeWarrior\bin\Plugins\Com`.

Listing 8.1 Sample HSST Host Program

```
#include "CodeWarriorCommands.h"
#include "HSSTInterface.h"
#include <cstdio>
#include <cstdlib>

unsigned __stdcall HSSTClientMain ( void *pArguments );

#define buf_size  1000          /* Data size */
```

```
/* Assigning name for Plugin and Menu Title */
extern const CWPluginID kToolbarTestPluginID =
    "HSST_host_sample";
extern const wchar_t* MenuTitle = L"HSST_host_sample";


unsigned __stdcall HSSTClientMain ( void *pArguments )
{
         IMWHSST_Client *pHSST = (IMWHSST_Client *)pArguments;

         long data[buf_size];
         size_t channel_1, channel_2, read_items, written_items;


         * Opening channel 1 and 2 from HOST side */
         HRESULT hr_1 = pHSST->hsst_open ( "channel_1",
           &channel_1 );
         HRESULT hr_2 = pHSST->hsst_open ( "channel_2",
           &channel_2 );


          /* HOST reading data from channel 1 */
     pHSST->hsst_read ( data, sizeof(long), buf_size, channel_1,
       &read_items );

     /* HOST writing data to channel 2 */
     pHSST->hsst_write( data, sizeof(long), buf_size, channel_2,
       &written_items );

     return 0;
}
```

Target Library Interface

This section describes the API calls for using High-Speed Simultaneous Transfer (HSST) from your target application.

At the end of this section, an example of a HSST target program is given (Listing 8.2 on page 266).

HSST_open

A target application uses this function to open a bidirectional communication channel with the host. The default setting is for the function to open an output channel in buffered mode. Opening a channel that has already been opened will result in the same channel ID being returned.

```
HSST_STREAM*  HSST_open  ( const char *stream );
```

`stream`

This parameter passes the communication channel name.

Returns This function returns the stream associated with the opened channel.

HSST_close

A target application uses this function to close a communication channel with the host.

```
int    HSST_close ( HSST_STREAM *stream );
```

`stream`

This parameter passes a pointer to the communication channel.

Returns This function returns `0` if the call was successful or `-1` if the call was unsuccessful.

HSST_setvbuf

A target application can use this function to perform the following actions:

- Set an open channel opened in write mode to use buffered mode

NOTE This can greatly improve performance.

- Resize the buffer in an existing buffered channel opened in write mode
- Provide an external buffer for an existing channel opened in write mode
- Reset buffering to unbuffered mode

You can use this function only after you successfully open the channel.

The contents of a buffer (either internal or external) at any time are indeterminate.

```
int    HSST_setvbuf (
    HSST_STREAM *rs,
    unsigned char *buf,
    int mode,
    size_t size );
```

`rs`

This parameter specifies a pointer to the communication channel.

`buf`

This parameter passes a pointer to an external buffer.

`mode`

This parameter passes the buffering mode as either buffered (specified as `HSSTFBUF`) or unbuffered (specified as `HSSTNBUF`).

`size`

This parameter passes the size of the buffer.

Returns This function returns `0` if the call was successful or `-1` if the call was unsuccessful.

NOTE You must flush the buffers before exiting the program to ensure that all the data that has been written is sent to the host. For more details, see HSST_flush.

HSST_write

A target application uses this function to write data for the host-side client application to read.

```
size_t   HSST_write (
    void *data,
    size_t size,
    size_t nmemb,
    HSST_STREAM *stream );
```

`data`

This parameter passes a pointer to the data buffer holding the data to write.

`size`

This parameter passes the size of the individual data elements to write.

`nmemb`

This parameter passes the number of data elements to write.

`stream`

This parameter passes a pointer to the communication channel.

Returns This function returns the number of data elements written.

HSST_read

A target application uses this function to read data sent by the host.

```
size_t   HSST_read  (
    void *data,
    size_t size,
    size_t nmemb,
    HSST_STREAM *stream );
```

`data`

This parameter passes a pointer to the data buffer into which to read the data.

`size`

This parameter passes the size of the individual data elements to read.

`nmemb`

This parameter passes the number of data elements to read.

`stream`

This parameter passes a pointer to the communication channel.

Returns This function returns the number of data elements read.

HSST_flush

A target application uses this function to flush out data buffered in a buffered output channel.

```
int HSST_flush ( HSST_STREAM *stream );
```

`stream`

This parameter passes a pointer to the communication channel. The High-Speed Simultaneous Transfer (HSST) feature flushes

all open buffered communication channels if this parameter is null.

Returns This function returns `0` if the call was successful or `-1` if the call was unsuccessful.

HSST_size

A target application uses this function to determine the size of unread data (in bytes) for the specified communication channel.

```
size_t HSST_size ( HSST_STREAM *stream );
```

`stream`

This parameter passes a pointer to the communication channel.

Returns This function returns the number of bytes of unread data.

HSST_raw_read

A target application uses this function to read raw data from a communication channel (without any automatic conversion for endianness while communicating).

```
size_t    HSST_raw_read  (
    void *ptr,
    size_t length,
    HSST_STREAM *rs );
```

`ptr`

This parameter specifies the pointer that points to the buffer into which data is read.

`length`

This parameter specifies the size of the buffer in bytes.

`rs`

This parameter specifies a pointer to the communication channel.

Returns This function returns the number of bytes of raw data read.

NOTE This function is useful for sending data structures (e.g., C-type structures).

HSST_raw_write

A target application uses this function to write raw data to a communication channel (without any automatic conversion for endianness while communicating).

```
size_t    HSST_raw_write (
    void *ptr,
    size_t length,
    HSST_STREAM *rs );
```

`ptr`

This parameter specifies the pointer that points to the buffer that holds the data to write.

`length`

This parameter specifies the size of the buffer in bytes.

`rs`

This parameter specifies a pointer to the communication channel.

Returns This function returns the number of data elements written.

NOTE This function is useful for sending data structures (e.g., C-type structures).

HSST_set_log_dir

A target application uses this function to set the host-side directory for storing temporary log files. Old logs that existed prior to the call to `HSST_set_log_dir()` are over-written. Logging stops when the channel is closed or when `HSST_set_log_dir()` is called with a null argument. These logs can be used by the host-side function `HSST_set_log_dir`.

```
int    HSST_set_log_dir (
    HSST_STREAM *stream,
    char *dir_name );
```

`stream`

This parameter passes a pointer to the communication channel.

`dir_name`

This parameter passes a pointer to the path to the directory in which to store temporary log files.

Returns This function returns `0` if the call was successful or `-1` if the call was unsuccessful.

HSST Target Program Example

In Listing 8.2 the HSST target program runs in parallel with the host plugin. The target communicates with the host-side (your computer).

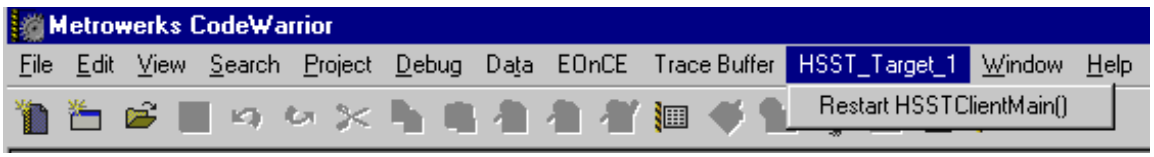
NOTE To restart the program after execution, click on **Restart HSST** as shown in Figure 8.1.

Listing 8.2 Sample HSST Target Program

```
#include <stdio.h>
#include <stdlib.h>
#include "HSST.h"
```

```
#define buf_size  1000     /* Data size */

long i, test_buffer[buf_size];

int main ( )
{
    HSST_STREAM *channel_1, *channel_2;
    int written_items=0;
    int read_items=0;

    for ( i = 0; i < buf_size; ++ i )
    {
         test_buffer[i] = i;
    }


    /* Opening channel 1 and 2 from TARGET side */
    channel_1 = HSST_open ( "channel_1" );
    channel_2 = HSST_open ( "channel_2" );


    /* TARGET writing data to channel 1 */
    written_items = HSST_write(test_buffer, sizeof(long),
        buf_size, channel_1);

    /* TARGET reading data from channel 2 */
    read_items = HSST_read(test_buffer, sizeof(long), buf_size,
        channel_2);

    return 0;
}
```

Figure 8.1 Restart HSST


Metrowerks CodeWarrior
File Edit View Search Project Debug Data EOnCE Trace Buffer HSST_Target_1 Window Help
Restart HSSTClientMain()

9

ELF Linker and Command Language

The CodeWarrior™ Executable and Linking Format (ELF) Linker makes a program file out of the object files of your project. The linker also allows you to manipulate code in different ways. You can define variables during linking, control the link order to the granularity of a single function, change the alignment, and even compress code and data segments so that they occupy less space in the output file.

All of these functions are accessed through commands in the linker command file (LCF). The linker command file has its own language complete with keywords, directives, and expressions, that are used to create the specifications for your output code. The syntax and structure of the linker command file is similar to that of a programming language.

This chapter contains the following sections:

- Structure of Linker Command Files
- Linker Command File Syntax
- Linker Command File Keyword Listing
- DSP56800E Command-Line Tools

Structure of Linker Command Files

Linker command files contain three main segments:

- Memory Segment
- Closure Blocks
- Sections Segment

A command file must contain a memory segment and a sections segment. Closure segments are optional.

Memory Segment

In the memory segment, available memory is divided into segments. The memory segment format looks like Listing 9.1.

Listing 9.1 Sample MEMORY Segment

```
MEMORY {
    segment_1 (RWX): ORIGIN = 0x8000, LENGTH = 0x1000
    segment_2 (RWX): ORIGIN = AFTER(segment_1), LENGTH = 0
    data      (RW) : ORIGIN = 0x2000, LENGTH = 0x0000
    #segment_name (RW) : ORIGIN = memory address, LENGTH = segment length
    #and so on...
   }
```

The first memory segment definition (`segment_1`) can be broken down as follows:

- the (`RWX`) portion of the segment definition pertains to the ELF access permission of the segment. The (RWX) flags imply **r**ead, **w**rite, and **e**xecute access.
- `ORIGIN` represents the start address of the memory segment (in this case `0x8000`).
- `LENGTH` represents the size of the memory segment (in this case `0x1000`).

Memory segments with RWX attributes are placed in to P: memory while RW attributes are placed into X: memory.

If you cannot predict how much space a segment will occupy, you can use the function `AFTER` and `LENGTH = 0` (unlimited length) to fill in the unknown values.

Closure Blocks

The linker is very good at deadstripping unused code and data. Sometimes, however, symbols need to be kept in the output file even if they are never directly referenced. Interrupt handlers, for example, are usually linked at special addresses, without any explicit jumps to transfer control to these places.

Closure blocks provide a way to make symbols immune from deadstripping. The closure is transitive, meaning that symbols referenced by the symbol being closed are also forced into closure, as are any symbols referenced by those symbols, and so on.

NOTE The closure blocks need to be in place before the SECTIONS definition in the linker command file.

The two types of closure blocks available are:

- **Symbol-level**

  Use `FORCE_ACTIVE` to include a symbol into the link that would not be otherwise included. An example is shown in Listing 9.2.

Listing 9.2 Sample Symbol-level Closure Block

```
FORCE_ACTIVE {break_handler, interrupt_handler, my_function}
```

- **Section-level**

  Use `KEEP_SECTION` when you want to keep a section (usually a user-defined section) in the link. Listing 9.3 shows an example.

Listing 9.3 Sample Section-level Closure Block

```
KEEP_SECTION {.interrupt1, .interrupt2}
```

A variant is `REF_INCLUDE`. It keeps a section in the link, but only if the file where it is coming from is referenced. This is very useful to include version numbers. Listing 9.4 shows an example of this.

Listing 9.4 Sample Section-level Closure Block With File Dependency

```
REF_INCLUDE {.version}
```

Sections Segment

Inside the sections segment, you define the contents of your memory segments, and define any global symbols to be used in the output file.

The format of a typical sections block looks like Listing 9.5.

NOTE As shown in Listing 9.5, the `.bss` section needs to go after the `.data` section.

Listing 9.5 Sample SECTIONS Segment

```
SECTIONS {
   .section_name : #the section name is for your reference
   {               #the section name must begin with a '.'
     filename.c  (.text) #put the .text section from filename.c
     filename2.c (.text) #then the .text section from filename2.c
     filename.c  (.data)
     filename2.c (.data)
     filename.c  (.bss)
     filename2.c (.bss)
      . = ALIGN (0x10);  #align next section on 16-byte boundary.
   } > segment_1    #this means "map these contents to segment_1"

   .next_section_name:
   {
      more content descriptions
   } > segment_x      # end of .next_section_name definition
   }                     # end of the sections block
```

Linker Command File Syntax

This section explains some practical ways in which to use the commands of the linker command file to perform common tasks.

Alignment

To align data on a specific byte-boundary, use the ALIGN and ALIGNALL commands to bump the location counter to the preferred boundary. For example, the following fragment uses `ALIGN` to bump the location counter to the next 16-byte boundary. An example is given in Listing 9.6.

Listing 9.6 Sample ALIGN Command Usage

```
file.c (.text)
. = ALIGN (0x10);
file.c (.data)    # aligned on a 16-byte boundary.
```

You can also align data on a specific byte-boundary with `ALIGNALL`, as shown in Listing 9.7.

Listing 9.7 Sample ALIGNALL Command Usage

```
file.c (.text)
ALIGNALL (0x10);  #everything past this point aligned on 16 bytes
file.c (.data)
```

Arithmetic Operations

Standard C arithmetic and logical operations may be used to define and use symbols in the linker command file. Table 9.1 shows the order of precedence for each operator. All operators are left-associative.

Table 9.1 Arithmetic Operators

Precedence	Operators
highest (1)	- ~ !
2	* / %
3	+ -
4	>> <<
5	== != > < <= >=
6	&
7	|
8	&&
9	||

Comments

Comments may be added by using the pound character (#) or C++ style double-slashes (//). C-style comments are not accepted by the LCF parser. Listing 9.8 shows examples of valid comments.

Listing 9.8 Sample Comments

```
#  This is a one-line comment
* (.text) // This is a partial-line comment
```

Deadstrip Prevention

The M56800E linker removes unused code and data from the output file. This process is called deadstripping. To prevent the linker from deadstripping unreferenced code and data, use the FORCE_ACTIVE, KEEP_SECTION, and REF_INCLUDE directives to preserve them in the output file.

Variables, Expressions, and Integral Types

This section explains variables, expressions, and integral types.

Variables and Symbols

All symbol names within a Linker Command File (LCF) start with the underscore character (_), followed by letters, digits, or underscore characters. Listing 9.9 shows examples of valid lines for a command file:

Listing 9.9 Valid Command File Lines

```
_dec_num = 99999999;
_hex_num_ = 0x9011276;
```

Variables that are defined within a `SECTIONS` section can only be used within a `SECTIONS` section in a linker command file.

Global Variables

Global variables are accessed in a linker command file with an 'F' prepended to the symbol name. This is because the compiler adds an 'F' prefix to externally defined symbols.

Listing 9.10 shows an example of using a global variable in a linker command file. This example sets the global variable `_foot`, declared in C with the `extern` keyword, to the location of the address location current counter.

Listing 9.10 Using a Global Variable in the LCF

```
F_foot = .;
```

If you use a global symbol in an LCF, as in Listing 9.10, you can access it from C program sources as shown in Listing 9.11.

Listing 9.11 Accessing a Global Symbol From C Program Sources

```
extern unsigned long _foot;
void main( void ) {
  unsigned long i;
  // ...
  i = _foot;   // _foot value determined in LCF
  // ...
}
```

Expressions and Assignments

You can create symbols and assign addresses to those symbols by using the standard assignment operator. An assignment may only be used at the start of an expression, and a semicolon is required at the end of an assignment statement. An example of standard assignment operator usage is shown in Listing 9.12.

Listing 9.12 Standard Assignment Operator Usage

```
_symbolicname = some_expression;# Legal
_sym1 + _sym2 = _sym3;          # ILLEGAL!
```

When an expression is evaluated and assigned to a variable, it is given either an absolute or a relocatable type. An absolute expression type is one in which the symbol contains the value that it will have in the output file. A relocatable expression is one in which the value is expressed as a fixed offset from the base of a section.

Integral Types

The syntax for linker command file expressions is very similar to the syntax of the C programming language. All integer types are `long` or `unsigned long`.

Octal integers (commonly know as base eight integers) are specified with a leading zero, followed by numeral in the range of zero through seven. Listing 9.13 shows valid octal patterns that you can put into your linker command file.

Listing 9.13 Sample Octal Patterns

```
_octal_number  = 012;
_octal_number2 = 03245;
```

Decimal integers are specified as a non-zero numeral, followed by numerals in the range of zero through nine. To create a negative integer, use the minus sign (-) in front of the number. Listing 9.14 shows examples of valid decimal integers that you can write into your linker command file.

Listing 9.14 Sample Decimal Integers

```
_dec_num       = 9999;
_decimalNumber = -1234;
```

Hexadecimal (base sixteen) integers are specified as `0x` or `0X` (a zero with an X), followed by numerals in the range of zero through nine, and/or characters `A` through `F`. Examples of valid hexadecimal integers that you can put in your linker command file appear in Listing 9.15.

Listing 9.15 Sample Hex Integers

```
_somenumber        = 0x0F21;
_fudgefactorspace = 0XF00D;
_hexonyou          = 0xcafe;
```

File Selection

When defining the contents of a `SECTION` block, specify the source files that are contributing to their sections. The standard method is listing the files as shown in Listing 9.16.

In a large project, the list can become very long. For this reason, you have to use the asterisk (*) keyword. The * keyword represents the filenames of every file in your project. Note that since you have already added the `.text` sections from the `main.c`, `file2.c`, and `file3.c files`, the `*` keyword does not allow addition of the `.text` sections from those files again.

Sometimes you may only want to include the files from a particular file group. The `GROUP` keyword allows you to specify all the files of a named file group.

Listing 9.16 Sample File Listing

```
SECTIONS {
  .example_section :
    {
       main.c  (.text)
       file2.c (.text)
       file3.c (.text)
       *  (.text)
       GROUP(fileGroup1) (.text)
       GROUP(fileGroup1) (.data)
     } > MYSEGMENT
   }
```

Function Selection

The OBJECT keyword allows precise control over how functions are placed within a section. For example, if the functions `pad` and `foot are` to be placed before anything else in a section, use the code as shown in the example in Listing 9.17.

Listing 9.17 Sample Function Selection Using OBJECT Keyword

```
SECTIONS {
  .program_section :
    {
      OBJECT (Fpad, main.c)
      OBJECT (Ffoot, main.c)
      * (.text)
     } > ROOT
  }
```

NOTE If an object is written once using the `OBJECT` function selection keyword, the same object will not be written again if you use the '`*`' file selection keyword.

ROM to RAM Copying

In embedded programming, it is common to copy a portion of a program resident in ROM into RAM at runtime. For example, program variables cannot be accessed until they are copied to RAM.

To indicate data or code that is meant to be copied from ROM to RAM, the data or code is assigned two addresses. One address is its resident location in ROM (where it is downloaded). The other is its intended location in RAM (where it is later copied in C code).

Use the `MEMORY` segment to specify the intended RAM location, and the `AT(address)` parameter to specify the resident ROM address.

For example, you have a program and you want to copy all your initialized data into RAM at runtime. Listing 9.18 shows the LCF you use to set up for writing data to ROM.

Listing 9.18 LCF to Setup for ROM to RAM Copy

```
MEMORY {
   .text (RWX) : ORIGIN = 0x8000, LENGTH = 0x0    # code (p:)
   .data (RW)  : ORIGIN = 0x3000, LENGTH = 0x0    # data (x:)-> RAM
}

SECTIONS{

     .main_application :
   {
      # .text sections

      *(.text)
      *(.rtlib.text)
      *(.fp_engine.txt)
      *(user.text)
   } > .text

    __ROM_Address = 0x2000
    .data : AT(__ROM_Address)  # ROM Address definition
   {
      # .data sections
      F__Begin_Data = .;       # Start location for RAM (0x3000)
      *(.data)                 # Write data to the section (ROM)
      *(fp_state.data);
      *(rtlib.data);
      F__End_Data = .;         # Get end location for RAM

      # .bss sections
      * (rtlib.bss.lo)
      * (.bss)
      F__ROM_Address = __ROM_Address

} > .data
}
```

To make the runtime copy from ROM to RAM, you need to know where the data starts in ROM (`__ROM_Address`) and the size of the block in ROM you want to copy to RAM. In the following example (Listing 9.19), copy all variables in the data section from ROM to RAM in C code.

Listing 9.19 ROM to RAM Copy From C After Writing Data Flash

```
#include <stdio.h>
#include <string.h>

int GlobalFlash = 6;

// From linker command file
extern __Begin_Data, __ROMAddress, __End_Data;

void main( void )
{
   unsigned short a = 0, b = 0, c = 0;
   unsigned long dataLen  = 0x0;
   unsigned short __myArray[] = { 0xdead, 0xbeef, 0xcafe };

   // Calculate the data length of the X: memory written to Flash
   dataLen = (unsigned long)&__End_Data -
   unsigned long)&__Begin_Data;

   // Block move from ROM to RAM
   memcpy( (unsigned long *)&__Begin_Data,
  (const unsigned long *)&__ROMAddress,dataLen );

   a = GlobalFlash;

   return;
 }
```

Stack and Heap

To reserve space for the stack and heap, arithmetic operations are performed to set the values of the symbols used by the runtime.

The Linker Command File (LCF) performs all the necessary stack and heap initialization. When Stationery is used to create a new project, the appropriate LCFs are added to the new project.

See any Stationery-generated LCFs for examples of how stack and heap are initialized.

Writing Data Directly to Memory

You can write data directly to memory using the `WRITEx` command in the linker command file. The `WRITEB` command writes a byte, the `WRITEH` command writes two bytes, and the `WRITEW` command writes four bytes. You insert the data at the section's current address.

Listing 9.20 Embedding Data Directly Into Output

```
.example_data_section :
{
   WRITEB 0x48;  //  'H'
   WRITEB 0x69;  //  'i'
   WRITEB 0x21;  //  '!'
}
```

Linker Command File Keyword Listing

This section explains the keywords available for use when creating CodeWarrior for DSP56800E application objects with the linker command file. Valid linker command file functions, keywords, directives, and commands are:

. (location counter)

Definition

The period character (`.`) always maintains the current position of the output location. Since the period always refers to a location in a `SECTIONS` block, it can not be used outside a section definition.

A period may appear anywhere a symbol is allowed. Assigning a value to period that is greater than its current value causes the location counter to move, but the location counter can never be decremented.

This effect can be used to create empty space in an output section. In the example below, the location counter is moved to a position that is `0x1000` bytes past the symbol `FSTART_`.

Example

```
.data :
{
      *(.data)
      *(.bss)
      FSTART_ = .;
      . = FSTART_ + 0x1000;
      __end = .;
} > DATA
```

ADDR

Definition — The `ADDR` function returns the address of the named section or memory segment.

Prototype — `ADDR (`*`sectionName`* | *`segmentName`* | *`symbol`*`)`

In the example below, `ADDR` is used to assign the address of `ROOT` to the symbol `__rootbasecode`.

Example

```
MEMORY{
     ROOT   (RWX) : ORIGIN = 0x8000, LENGTH = 0
}

SECTIONS{
  .code :
  {
  __rootbasecode = ADDR(ROOT);
     *(.text);
  } > ROOT
}
```

NOTE In order to use segmentName with this command, the segmentName must start with the period character even though segmentNames are not required to start with the period character by the linker, as is the case with sectionName.

ALIGN

Definition — The `ALIGN` function returns the value of the location counter aligned on a boundary specified by the value of `alignValue`. The `alignValue` must be a power of two.

Prototype — `ALIGN(alignValue)`

Note that `ALIGN` does not update the location counter; it only performs arithmetic. To update the location counter, use an assignment such as:

Example

```
. = ALIGN(0x10);     #update location counter to 16
                      #byte alignment
```

ALIGNALL

Definition

`ALIGNALL` is the command version of the `ALIGN` function. It forces the minimum alignment for all the objects in the current segment to the value of `alignValue`. The `alignValue` must be a power of two.

Prototype

```
ALIGNALL(alignValue);
```

Unlike its counterpart ALIGN, `ALIGNALL` is an actual command. It updates the location counter as each object is written to the output.

Example

```
.code :
{
  ALIGNALL(16);  // Align code on 16 byte boundary
  *     (.init)
  *     (.text)

  ALIGNALL(16);  //align data on 16 byte boundary
  *     (.rodata)
} > .text
```

FORCE_ACTIVE

Definition

The `FORCE_ACTIVE` directive allows you to specify symbols that you do not want the linker to deadstrip. You must specify the symbol(s) you want to keep before you use the SECTIONS keyword.

Prototype

`FORCE_ACTIVE{ symbol[, symbol] }`

GROUP

Definition — The `GROUP` keyword allows you to selectively include files and sections from certain file groups.

Prototype — `GROUP (`*`fileGroup`*`) (`*`sectionType`*`)`

Definition — In the example, all the .bss sections of the files in the file group named `PAD` are specified.

Example — `GROUP (PAD) (.bss)`

INCLUDE

Definition — The `INCLUDE` command allows you to include a binary file in the output file.

Prototype — `INCLUDE` *`filename`*

KEEP_SECTION

Definition — The `KEEP_SECTION` directive allows you to specify sections that you do not want the linker to deadstrip. You must specify the section(s) you want to keep before you use the SECTIONS keyword.

Prototype — `KEEP_SECTION{` *`sectionType`*`[,` *`sectionType`*`] }`

MEMORY

Definition

The `MEMORY` directive allows you to describe the location and size of memory segment blocks in the target. This directive specifies the linker the memory areas to avoid, and the memory areas into which it links the code and date.

The linker command file may only contain one `MEMORY` directive. However, within the confines of the `MEMORY` directive, you may define as many memory segments as you wish.

Prototype

```
MEMORY { memory_spec }
```

The `memory_spec` is:

segmentName (*accessFlags*) : ORIGIN = *address,* LENGTH = *length,* [COMPRESS] [> *fileName*]

`segmentName` can include alphanumeric characters and underscore '_' characters.

`accessFlags` are passed into the output ELF file (`Phdr.p_flags`). The `accessFlags` can be:

- R-read
- W-write
- X-executable (for P: memory placement)

`ORIGIN address` is one of the following:

a memory address — Specify a hex address, such as `0x8000`.

an AFTER command — Use the `AFTER(name [,name])` command to tell the linker to place the memory segment after the specified segment. In the example below, `overlay1` and `overlay2` are placed after the code segment. When multiple memory segments are specified as parameters for `AFTER`, the highest memory address is used.

Example

```
memory{
code     (RWX)   : ORIGIN = 0x8000,       LENGTH = 0
overlay1 (RWX)   : ORIGIN = AFTER(code),  LENGTH = 0
overlay2 (RWX)   : ORIGIN = AFTER(code),  LENGTH = 0
data     (RW)    : ORIGIN = 0x1000,       LENGTH = 0
}
```

`ORIGIN` is the assigned address.

`LENGTH` is one of the following:

a value greater than zero — If you try to put more code and data into a memory segment than your specified length allows, the linker stops with an error.

autolength by specifying zero — When the length is 0, the linker lets you put as much code and data into a memory segment as you want.

NOTE There is no overflow checking with autolength. The linker can produce an unexpected result if you use the autolength feature without leaving enough free memory space to contain the memory segment. For this reason, when you use autolength, use the `AFTER` keyword to specify origin addresses.

`> fileName` is an option to write the segment to a binary file on disk instead of an ELF program header. The binary file is put in the same folder as the ELF output file. This option has two variants:

`>fileName`	Writes the segment to a new file.
`>>fileName`	Appends the segment to an existing file.

OBJECT

Definition — The `OBJECT` keyword allows control over the order in which functions are placed in the output file.

Prototype — `OBJECT (function, sourcefile.c)`

It is important to note that if you write an object to the output file using the `OBJECT` keyword, the same object will not be written again by either the `GROUP` keyword or the '*' wildcard.

REF_INCLUDE

Definition — The `REF_INCLUDE` directive allows you to specify sections that you do not want the linker to deadstrip, but only if they satisfy a certain condition: the file that contains the section must be referenced. This is useful if you want to include version information from your source file components. You must specify the section(s) you want to keep before you use the SECTIONS keyword.

Prototype — `REF_INCLUDE{ sectionType [, sectionType]}`

SECTIONS

Definition — A basic `SECTIONS` directive has the following form:

Prototype — `SECTIONS { <section_spec> }`

`section_spec` is one of the following:

- *sectionName*: [AT (*loadAddress*)] {contents} > *segmentName*
- *sectionName*: [AT (*loadAddress*]] {contents} >> *segmentName*

`sectionName` is the section name for the output section. It must start with a period character. For example, "`.mysection`".

`AT (loadAddress)` is an optional parameter that specifies the address of the section. The default (if not specified) is to make the load address the same as the relocation address.

`contents` are made up of statements. These statements can:

- Assign a value to a symbol.
- Describe the placement of an output section, including which input sections are placed into it.

`segmentName` is the predefined memory segment into which you want to put the contents of the section. The two variants are:

`>segmentName`	Places the section contents at the beginning of the memory segment `segmentName`.
`>>segmentName`	Appends the section contents to the memory segment `segmentName`.

Here is an example section definition:

Example

```
SECTIONS {
  .text : {
            F_textSegmentStart = .;
            footpad.c (.text)
            . = ALIGN (0x10);
            padfoot.c (.text)
            F_textSegmentEnd = .;
  } > TEXT
  .data : { *(.data) } > DATA
  .bss  : { *(.bss) > BSS
            *(COMMON)
  }
}
```

SIZEOF

Definition — The `SIZEOF` function returns the size of the given segment or section. The return value is the size in bytes.

Prototype — `SIZEOF(`*`sectionName`* | *`segmentName`* | *`symbol`*`)`

NOTE — In order to use segmentName with this command, the segmentName must start with the period character even though segmentNames are not required to start with the period character by the linker, as is the case with sectionName.

SIZEOFW

Definition — The `SIZEOFW` function returns the size of the given segment or section. The return value is the size in words.

Prototype — `SIZEOFW(`*`sectionName`* | *`segmentName`* | *`symbol`*`)`

In order to use segmentName with this command, the segmentName must start with the period character even though segmentNames are not required to start with the period character by the linker, as is the case with sectionName.

WRITEB

Definition — The WRITEB command inserts a byte of data at the current address of a section.

Prototype — `WRITEB (expression);`

`expression` is any expression that returns a value `0x00` to `0xFF`.

WRITEH

Definition — The WRITEH command inserts two bytes of data at the current address of a section.

Prototype — `WRITEH (expression);`

`expression` is any expression that returns a value `0x0000` to `0xFFFF`.

WRITEW

Definition — The WRITEW command inserts 4 bytes of data at the current address of a section.

Prototype — `WRITEW (expression);`

`expression` is any expression that returns a value `0x00000000` to `0xFFFFFFFF`.

DSP56800E Command-Line Tools

This section contains the following topics:

- Usage
- Response File
- Sample Build Script
- Arguments

Usage

To call the command-line tools, use the following format:

Table 9.2 Format

Tools	File Names	Format
Compiler	mwcc56800e.exe	compiler-options [linker-options] file-list
Linker	mwld56800e.exe	linker-options file-list
Assembler	mwasm56800e.exe	assembler-options file-list

The compiler automatically calls the linker by default and any options from the linker is passed on by the compiler to the assembler. However, you may choose to only compile with the `-c` flag. In this case, the assembler will only assemble and will not call the linker.

Also, available are environment variables. These are used to provide path information for includes or libraries, and to specify which libraries are to be included. You can specify the variables listed in Table 9.3.

Table 9.3 Environment Variables

Tool	Library	Description
Compiler	**MWC56800EIncludes**	Similar to Access Paths panel; separate paths with ';' and prefix a path with '+' to specify a recursive path

Tool	Library	Description
Linker	**MW56800ELibraries**	Similar to MWC56800EIncludes
	MW56800ELibraryFiles	List of library names to link with project; separate with ';'
Assembler	**MWAsm56800EIncludes**	(similar to MWC56800EIncludes)

These are the target-specific variables, and will only work with the DSP56800E tools. The generic variables **MWCIncludes**, **MWLibraries**, **MWLibraryFiles**, and **MWAsmIncludes** apply to all target tools on your system (such as Windows). If you only have the DSP56800E tools installed, then you may use the generic variables if you prefer.

Response File

In addition to specifying commands in the argument list, you may also specify a "response file". A response file's filename begins with an '@' (for example, @file), and the contents of the response file are commands to be inserted into the argument list. The response file supports standard UNIX-style comments. For example, the response file @file, contain the following:

```
# Response file @file
-o out.elf        # change output file name to 'out.elf'
-g                # generate debugging symbols
```

The above response file can used in a command such as:

```
mwcc56800e @file main.c
```

It would be the same as using the following command:

```
mwcc56800e -o out.elf -g main.c
```

Sample Build Script

This following is a sample of a DOS batch (BAT) file. The sample demonstrates:

- Setting of the environmental variables.
- Using the compiler to compile and link a set of files.

```
REM *** set GUI compiler path ***
set COMPILER={path to compiler}

REM *** set includes path ***
set MWCIncludes=+%COMPILER%\M56800E Support
set MWLibraries=+%COMPILER%\M56800E Support
set MWLibraryFiles=Runtime 56800E.Lib;MSL C 56800E.lib

REM *** add CLT directory to PATH ***
set PATH=%PATH%;%COMPILER%\DSP56800E_EABI_Tools\Command_Line_Tools\

REM *** compile options and files ***
set COPTIONS=-O3
set CFILELIST=file1.c file2.c
set LOPTIONS=-m FSTART_ -o output.elf -g
set LCF=linker.cmd

REM *** compile, assemble and link ***
mwcc56800e %COPTIONS% %CFILELIST%
mwasm56800e %AFILELIST%
mwld56800e %LOPTIONS% %LFILELIST% %LCF%
```

Arguments

General Command-Line Options

```
--------------------------------------------------------------------------------
General Command-Line Options

   All the options are passed to the linker unless otherwise noted.

   Please see '-help usage' for details about the meaning of this help.
--------------------------------------------------------------------------------
  -help [keyword[,...]]     # global; for this tool;
                            #   display help
     usage                  #    show usage information
     [no]spaces             #    insert blank lines between options in
                            #      printout
     all                    #    show all standard options
     [no]normal             #    show only standard options
     [no]obsolete           #    show obsolete options
     [no]ignored            #    show ignored options
     [no]deprecated         #    show deprecated options
     [no]meaningless        #    show options meaningless for this target
     [no]compatible         #    show compatibility options
     opt[ion]=name          #    show help for a given option; for 'name',
                            #      maximum length 63 chars
     search=keyword         #    show help for an option whose name or help
                            #      contains 'keyword' (case-sensitive); for
                            #      'keyword', maximum length 63 chars
     group=keyword          #    show help for groups whose names contain
                            #      'keyword' (case-sensitive); for 'keyword',
                            #       maximum length 63 chars
     tool=keyword[,...]     #    categorize groups of options by tool;
                            #      default
        all                 #       show all options available in this tool
        this                #       show options executed by this tool;
                            #        default
        other|skipped       #       show options passed to another tool
        both                #       show options used in all tools
                            #
                            #
  -version                  # global; for this tool;
                            #   show version, configuration, and build date
  -timing                   # global; collect timing statistics
  -progress                 # global; show progress and version
  -v[erbose]                # global; verbose information; cumulative;
                            #   implies -progress
  -search                   # global; search access paths for source files
                            #   specified on the command line; may specify
                            #   object code and libraries as well; this
                            #   option provides the IDE's 'access paths'
```

```
                              #   functionality
  -[no]wraplines              # global; word wrap messages; default
  -maxerrors max              # specify maximum number of errors to print, zero
                              #   means no maximum; default is 0
  -maxwarnings max            # specify maximum number of warnings to print,
                              #   zero means no maximum; default is 0
  -msgstyle keyword           # global; set error/warning message style
     mpw                      #    use MPW message style
     std                      #    use standard message style; default
     gcc                      #    use GCC-like message style
     IDE                      #    use CW IDE-like message style
     parseable                #    use context-free machine-parseable message
                              #      style
                              #
  -[no]stderr                 # global; use separate stderr and stdout streams;
                              #   if using -nostderr, stderr goes to stdout
```

Compiler

```
-------------------------------------------------------------------------------
Preprocessing, Precompiling, and Input File Control Options
-------------------------------------------------------------------------------
  -c                          # global; compile only, do not link
  -[no]codegen                # global; generate object code
  -[no]convertpaths           # global; interpret #include filepaths specified
                              #   for a foreign operating system; i.e.,
                              #   <sys/stat.h> or <:sys:stat.h>; when enabled,
                              #   '/' and ':' will separate directories and
                              #   cannot be used in filenames (note: this is
                              #   not a problem on Win32, since these
                              #   characters are already disallowed in
                              #   filenames; it is safe to leave the option
                              #   'on'); default
  -cwd keyword                # specify #include searching semantics:  before
                              #   searching any access paths, the path
                              #   specified by this option will be searched
     proj                     #    begin search in current working directory;
                              #      default
     source                   #    begin search in directory of source file
     explicit                 #    no implicit directory; only search '-I' or
                              #      '-ir' paths
     include                  #    begin search in directory of referencing
                              #      file
                              #
  -D+ | -d[efine]             # cased; define symbol 'name' to 'value' if
    name[=value]              #   specified, else '1'
  -[no]defaults               # global; passed to linker;
                              #   same as '-[no]stdinc'; default
  -dis[assemble]              # global; passed to all tools;
```

```
                          #   disassemble files to stdout
  -E                      # global; cased; preprocess source files
  -EP                     # global; cased; preprocess and strip out #line
                          #   directives
  -ext extension          # global; specify extension for generated object
                          #   files; with a leading period ('.'), appends
                          #   extension; without, replaces source file's
                          #   extension; for 'extension', maximum length 14
                          #   chars; default is none
  -gccinc[ludes]          # global; adopt GCC #include semantics:  add '-I'
                          #   paths to system list if '-I-' is not
                          #   specified, and search directory of
                          #   referencing file first for #includes (same as
                          #   '-cwd include')
  -i- | -I-               # global; change target for '-I' access paths to
                          #   the system list; implies '-cwd explicit';
                          #   while compiling, user paths then system paths
                          #   are searched when using '#include "..."; only
                          #   system paths are searched with '#include
                          #   <...>'
  -I+ | -i path            # global; cased; append access path to current
                          #   #include list(see '-gccincludes' and '-I-')
  -ir path                # global; append a recursive access path to
                          #   current #include list
  -[no]keepobj[ects]      # global; keep object files generated after
                          #   invoking linker; if disabled, intermediate
                          #   object files are temporary and deleted after
                          #   link stage; objects are always kept when
                          #   compiling
  -M                      # global; cased; scan source files for
                          #   dependencies and emit Makefile, do not
                          #   generate object code
  -MM                     # global; cased; like -M, but do not list system
                          #   include files
  -MD                     # global; cased; like -M, but write dependency
                          #   map to a file and generate object code
  -MMD                    # global; cased; like -MD, but do not list system
                          #   include files
  -make                   # global; scan source files for dependencies and
                          #   emit Makefile, do not generate object code
  -nofail                 # continue working after errors in earlier files
  -nolink                 # global; compile only, do not link
  -noprecompile           # do not precompile any files based on the
                          #   filename extension
  -nosyspath              # global; treat #include <...> like #include
                          #   "..."; always search both user and system
                          #   path lists
  -o file|dir             # specify output filename or directory for object
                          #   file(s) or text output, or output filename
```

```
                              #   for linker if called
  -P                          # global; cased; preprocess and send output to
                              #   file; do not generate code
  -precompile file|dir        # generate precompiled header from source; write
                              #   header to 'file' if specified, or put header
                              #   in 'dir'; if argument is "", write header to
                              #   source-specified location; if neither is
                              #   defined, header filename is derived from
                              #   source filename; note: the driver can tell
                              #   whether to precompile a file based on its
                              #   extension; '-precompile file source' then is
                              #   the same as '-c -o file source'
  -preprocess                 # global; preprocess source files
  -prefix file                # prefix text file or precompiled header onto all
                              #   source files
  -S                          # global; cased; passed to all tools;
                              #   disassemble and send output to file
  -[no]stdinc                 # global; use standard system include paths
                              #   (specified by the environment variable
                              #   %MWCIncludes%); added after all system '-I'
                              #   paths; default
  -U+ | -u[ndefine] name      # cased; undefine symbol 'name'

-------------------------------------------------------------------------------
Front-End C/C++ Language Options
-------------------------------------------------------------------------------
  -ansi keyword               # specify ANSI conformance options, overriding
                              #   the given settings
     off                      #    same as '-stdkeywords off', '-enum min', and
                              #      '-strict off'; default
     on|relaxed               #    same as '-stdkeywords on', '-enum min', and
                              #      '-strict on'
     strict                   #    same as '-stdkeywords on', '-enum int', and
                              #      '-strict on'
                              #
  -ARM on|off                 # check code for ARM (Annotated C++ Reference
                              #   Manual) conformance; default is off
  -bool on|off                # enable C++ 'bool' type, 'true' and 'false'
                              #    constants; default is off
  -char keyword               # set sign of 'char'
     signed                   #    chars are signed; default
     unsigned                 #    chars are unsigned
                              #
  -Cpp_exceptions on|off      # passed to linker;
                              #    enable or disable C++ exceptions; default is
                              #    on
  -dialect | -lang keyword    # passed to linker;
                              #   specify source language
     c                        #    treat source as C always
```

```
   c++                      #    treat source as C++ always
   ec++                     #    generate warnings for use of C++ features
                            #    outside Embedded C++ subset (implies
                            #    'dialect cplus')
                            #      ‘dialect cplus’)
-enum keyword               # specify word size for enumeration types
   min                      #   use minimum sized enums; default
   int                      #   use int-sized enums
                            #
-inline keyword[,...]       # specify inline options
   on|smart                 #    turn on inlining for 'inline' functions;
                            #      default
   none|off                 #    turn off inlining
   auto                     #    auto-inline small functions (without
                            #      'inline' explicitly specified)
   noauto                   #    do not auto-inline; default
   all                      #    turn on aggressive inlining: same as
                            #      '-inline on, auto'
   deferred                 #    defer inlining until end of compilation
                            #      unit; this allows inlining of functions in
                            #    both directions
   level=n                  #    cased; inline functions up to 'n' levels
                            #      deep; level 0 is the same as '-inline on';
                            #      for 'n', range 0 - 8
                            #
-iso_templates on|off       # enable ISO C++ template parser (note: this
                            #   requires a different MSL C++ library);
                            #   default is off
-[no]mapcr                  # reverse mapping of '\n' and '\r' so that
                            #   '\n'==13 and '\r'==10 (for Macintosh MPW
                            #   compatability)
-msext keyword              # [dis]allow Microsoft VC++ extensions
   on                       #    enable extensions: redefining macros,
                            #      allowing XXX::yyy syntax when declaring
                            #      method yyy of class XXX,
                            #      allowing extra commas,
                            #      ignoring casts to the same type,
                            #      treating function types with equivalent
                            #      parameter lists but different return types
                            #      as equal,
                            #      allowing pointer-to-integer conversions,
                            #      and various syntactical differences
  off                       #    disable extensions; default on non-x86
                            #      targets
                            #
-[no]multibyte[aware]       # enable multi-byte character encodings for
                            #   source text, comments, and strings
-once                       # prevent header files from being processed more
                            #  than once
```

```
  -pragma                    # define a pragma for the compiler such as
                             #  "#pragma ..."
  -r[equireprotos]           # require prototypes
  -relax_pointers            # relax pointer type-checking rules
  -RTTI on|off               # select run-time typing information (for C++);
                             #   default is on
  -som                       # enable Apple's Direct-to-SOM implementation
  -som_env_check             # enables automatic SOM environment and new
                             #   allocation checking; implies -som
  -stdkeywords on|off        # allow only standard keywords; default is off
  -str[ings] keyword[,...]   # specify string constant options
     [no]reuse               #    reuse strings; equivalent strings are the
                             #      same object; default
     [no]pool                #    pool strings into a single data object
     [no]readonly            #    make all string constants read-only
                             #
  -strict on|off             # specify ANSI strictness checking; default is
                             #   off
  -trigraphs on|off          # enable recognition of trigraphs; default is off
  -wchar_t on|off            # enable wchar_t as a built-in C++ type; default
                             #   is on

-------------------------------------------------------------------------------
Optimizer Options

   Note that all options besides '-opt off|on|all|space|speed|level=...' are
   for backwards compatibility; other optimization options may be superceded
   by use of '-opt level=xxx'.
-------------------------------------------------------------------------------
  -O                         # same as '-O2'
  -O+keyword[,...]           # cased; control optimization; you may combine
                             #   options as in '-O4,p'
     0                       #    same as '-opt off'
     1                       #    same as '-opt level=1'
     2                       #    same as '-opt level=2'
     3                       #    same as '-opt level=3'
     4                       #    same as '-opt level=4'
     p                       #    same as '-opt speed'
     s                       #    same as '-opt space'
                             #
  -opt keyword[,...]         # specify optimization options
     off|none                #    suppress all optimizations; default
     on                      #    same as '-opt level=2'
     all|full                #    same as '-opt speed, level=4'
     [no]space               #    optimize for space
     [no]speed               #    optimize for speed
     l[evel]=num             #    set optimization level:
                             #      level 0: no optimizations
                             #
```

```
                            #      level 1: global register allocation,
                            #      peephole, dead code elimination
                            #
                            #      level 2: adds common subexpression
                            #      elimination and copy propagation
                            #
                            #      level 3: adds loop transformations,
                            #      strength reduction, loop-invariant code
                            #      motion
                            #
                            #      level 4: adds repeated common
                            #      subexpression elimination and
                            #      loop-invariant code motion
                            #      ; for 'num', range 0 - 4; default is 0
    [no]cse                 #    common subexpression elimination
      [no]commonsubs        #
    [no]deadcode            #    removal of dead code
    [no]deadstore           #    removal of dead assignments
    [no]lifetimes           #    computation of variable lifetimes
    [no]loop[invariants]    #    removal of loop invariants
    [no]prop[agation]       #    propagation of constant and copy assignments
    [no]strength            #    strength reduction; reducing multiplication
                            #      by an index variable into addition
    [no]dead                #    same as '-opt [no]deadcode' and '-opt
                            #      [no]deadstore'
    display|dump            #    display complete list of active
                            #      optimizations
                            #

------------------------------------------------------------------------------
DSP M56800E CodeGen Options
------------------------------------------------------------------------------
  -DO keyword               # for this tool;
                            #   specify hardware DO loops
    off                     #    no hardware DO loops; default
    nonested                #    hardware DO loops but no nested ones
    nested                  #    nested hardware DO loops
                            #
  -padpipe                  # for this tool;
                            #   pad pipeline for debugger
  -ldata | -largedata       # for this tool;
                            #  data space not limited to 64K
  -globalsInLowerMemory     # for this tool;
                            #   globals live in lower memory; implies '-large
                            #   data model'
  -sprog | -smallprog       # for this tool;
                            #   program space limited to 64K

------------------------------------------------------------------------------
```

```
Debugging Control Options
-------------------------------------------------------------------------------
  -g                        # global; cased; generate debugging information;
                            #   same as '-sym full'
  -sym keyword[,...]        # global; specify debugging options
     off                    #    do not generate debugging information;
                            #      default
     on                     #    turn on debugging information
     full[path]             #    store full paths to source files
                            #
-------------------------------------------------------------------------------
C/C++ Warning Options
-------------------------------------------------------------------------------
  -w[arn[ings]]             # global; for this tool;
    keyword[,...]           #   warning options
     off                    #    passed to all tools;
                            #      turn off all warnings
     on                     #    passed to all tools;
                            #      turn on most warnings
     [no]cmdline            #    passed to all tools;
                            #      command-line driver/parser warnings
     [no]err[or] |          #    passed to all tools;
       [no]iserr[or]        #      treat warnings as errors
     all                    #    turn on all warnings, require prototypes
     [no]pragmas |          #    illegal #pragmas
       [no]illpragmas       #
     [no]empty[decl]        #    empty declarations
     [no]possible |         #    possible unwanted effects
       [no]unwanted         #
     [no]unusedarg          #    unused arguments
     [no]unusedvar          #    unused variables
     [no]unused             #    same as -w [no]unusedarg,[no]unusedvar
     [no]extracomma |       #    extra commas
       [no]comma            #
     [no]pedantic |         #    pedantic error checking
       [no]extended         #
     [no]hidevirtual |      #    hidden virtual functions
       [no]hidden[virtual]  #
     [no]implicit[conv]     #    implicit arithmetic conversions
     [no]notinlined         #    'inline' functions not inlined
     [no]largeargs          #    passing large arguments to unprototyped
                            #      functions
     [no]structclass        #    inconsistent use of 'class' and 'struct'
     [no]padding            #    padding added between struct members
     [no]notused            #    result of non-void-returning function not
                            #      used
     [no]unusedexpr         #    use of expressions as statements without
                            #      side effects
     [no]ptrintconv         #    conversions from pointers to integers, and
```

```
                            #      vice versa
     display|dump           #    display list of active warnings
                            #
```

Linker

```
-------------------------------------------------------------------------------
Command-Line Linker Options
-------------------------------------------------------------------------------
  -dis[assemble]            # global; disassemble object code and do not
                            #   link; implies '-nostdlib'
  -L+ | -l path             # global; cased; add library search path; default
                            #   is to search current working directory and
                            #   then system directories (see '-defaults');
                            #   search paths have global scope over the
                            #   command line and are searched in the order
                            #   given
  -lr path                  # global; like '-l', but add recursive library
                            #   search path
  -l+file                   # cased; add a library by searching access paths
                            #   for file named lib<file>.<ext> where <ext> is
                            #   a typical library extension; added before
                            #   system libraries (see '-defaults')
  -[no]defaults             # global; same as -[no]stdlib; default
  -nofail                   # continue importing or disassembling after
                            #   errors in earlier files
  -[no]stdlib               # global; use system library access paths
                            #   (specified by %MWLibraries%) and add system
                            #   libraries (specified by %MWLibraryFiles%);
                            #   default
  -S                        # global; cased; disassemble and send output to
                            #   file; do not link; implies '-nostdlib'

-------------------------------------------------------------------------------
ELF Linker Options
-------------------------------------------------------------------------------
  -[no]dead[strip]          # enable dead-stripping of unused code; default
  -force_active             # specify a list of symbols as undefined; useful
    symbol[,...]            #   to force linking of static libraries
                            #
  -keep[local] on|off       # keep local symbols (such as relocations and
                            #   output segment names) generated during link;
                            #   default is on
  -m[ain] symbol            # set main entry point for application or shared
                            #   library; use '-main ""' to specify no entry
                            #   point; for 'symbol', maximum length 63 chars;
                            #   default is 'FSTART_'
  -map [keyword[,...]]      # generate link map file
     closure                #    calculate symbol closures
     unused                 #    list unused symbols
```

```
                              #
  -sortbyaddr                 # sort S-records by address; implies '-srec'
  -srec                       # generate an S-record file; ignored when
                              #   generating static libraries
  -sreceol keyword            # set end-of-line separator for S-record file;
                              #   implies '-srec'
     mac                      #    Macintosh ('\r')
     dos                      #    DOS ('\r\n'); default
     unix                     #    Unix ('\n')
                              #
  -sreclength length          # specify length of S-records (should be a
                              #   multiple of 4); implies '-srec'; for
                              #   'length', range 8 - 252; default is 64
  -usebyteaddr                # use byte address in S-record file; implies
                              #   '-srec'
  -o file                     # specify output filename

-------------------------------------------------------------------------------
DSP M56800E Project Options
-------------------------------------------------------------------------------
  -application                # global; generate an application; default
  -library                    # global; generate a static library

-------------------------------------------------------------------------------
DSP M56800E CodeGen Options
-------------------------------------------------------------------------------
  -ldata | -largedata         # data space not limited to 64K

-------------------------------------------------------------------------------
Linker C/C++ Support Options
-------------------------------------------------------------------------------
  -Cpp_exceptions on|off      # enable or disable C++ exceptions; default is on
  -dialect | -lang keyword    # specify source language
     c                        #    treat source as C++ unless its extension is
                              #      '.c', '.h', or '.pch'; default
     c++                      #    treat source as C++ always
                              #
```

```
-------------------------------------------------------------------------
Debugging Control Options
-------------------------------------------------------------------------
  -g                        # global; cased; generate debugging information;
                            #   same as '-sym full'
  -sym keyword[,...]        # global; specify debugging options
     off                    #    do not generate debugging information;
                            #      default
     on                     #    turn on debugging information
     full[path]             #    store full paths to source files
                            #

-------------------------------------------------------------------------
Warning Options
-------------------------------------------------------------------------
  -w[arn[ings]]             # global; warning options
    keyword[,...]           #
     off                    #    turn off all warnings
     on                     #    turn on all warnings
     [no]cmdline            #    command-line parser warnings
     [no]err[or] |          #    treat warnings as errors
       [no]iserr[or]        #
     display|dump           #    display list of active warnings
                            #

-------------------------------------------------------------------------
ELF Disassembler Options
-------------------------------------------------------------------------
  -show keyword[,...]       # specify disassembly options
     only|none              #    as in '-show none' or, e.g.,
                            #      '-show only,code,data'
     all                    #    show everything; default
     [no]code | [no]text    #    show disassembly of code sections; default
     [no]comments           #    show comment field in code; implies '-show
                            #      code'; default
     [no]extended           #    show extended mnemonics; implies '-show
                            #      code'; default
     [no]data               #    show data; with '-show verbose', show hex
                            #      dumps of sections; default
     [no]debug | [no]sym    #    show symbolics information; default
     [no]exceptions         #    show exception tables; implies '-show data';
                            #      default
     [no]headers            #    show ELF headers; default
     [no]hex                #    show addresses and opcodes in code
                            #      disassembly; implies '-show code'; default
     [no]names              #    show symbol table; default
     [no]relocs             #    show resolved relocations in code and
                            #      relocation tables; default
     [no]source             #    show source in disassembly; implies '-show
```

```
                          #      code'; with '-show verbose', displays
                          #      entire source file in output, else shows
                          #      only four lines around each function;
                          #      default
     [no]xtables          #    show exception tables; default
     [no]verbose          #    show verbose information, including hex dump
                          #      of program segments in applications;
                          #      default
                          #
```

Assembler

```
-------------------------------------------------------------------------------
Assembler Control Options
-------------------------------------------------------------------------------
  -[no]case                # identifiers are case-sensitive; default
  -[no]debug               # generate debug information
  -[no]macro_expand        # expand macro in listin output
  -[no]assert_nop          # add nop to resolve pipeline dependency; default
  -[no]warn_nop            # emit warning when there is a pipeline
                           #   dependency
  -[no]warn_stall          # emit warning when there is a hardware stall
  -[no]legacy              # allow legacy DSP56800 instructions(imply
                           #   data/prog 16)
  -[no]debug_workaround     # Pad nop workaround debuggin issue in some
                           #   implementation; default
  -data keyword            # data memory compatibility
     16                    #    16 bit; default
     24                    #    24 bit
                           #
  -prog keyword            # program memory compatibility
     16                    #    16 bit; default
     19                    #    19 bit
     21                    #    21 bit
                           #
-------------------------------------------------------------------------------
```

10

Libraries and Runtime Code

You can use a variety of libraries with the CodeWarrior™ IDE. The libraries include ANSI-standard libraries for C, runtime libraries, and other codes. This chapter explains how to use these libraries for DSP56800E development.

With respect to the Metrowerks Standard Library (MSL) for C, this chapter is an extension of the *MSL C Reference*. Consult that manual for general details on the standard libraries and their functions.

This chapter contains the following sections:

- MSL for DSP56800E
- Runtime Initialization
- EOnCE Library

MSL for DSP56800E

This section explains the Metrowerks Standard Library (MSL) that has been modified for use with DSP56800E.

Using MSL for DSP56800E

CodeWarrior for DSP56800E includes a version of the Metrowerks Standard Library (MSL). MSL is a complete C library for use in embedded projects. All of the sources necessary to build MSL are included in CodeWarrior for DSP56800E, along with the project files for different configurations of MSL. If you already have a version of the CodeWarrior IDE installed on your computer, the CodeWarrior installer adds the new files needed for building versions of MSL for DSP56800E.

The project directory for the DSP56800E MSL is:

```
CodeWarrior\M56800E Support\msl\MSL_C\DSP_56800E\projects\MSL C 56800E.mcp
```

Do not modify any of the source files included with MSL. If you need to make changes based on your memory configuration, make changes to the runtime libraries.

Ensure that you include one or more of the header files located in the following directory:

```
CodeWarrior\M56800E Support\msl\MSL_C\DSP_56800E\inc
```

When you add the relative-to-compiler path to your project, the appropriate MSL and runtime files will be found by your project. If you create your project from Stationery, the new project will have the proper support access path.

Console and File I/O

DSP56800E Support provides standard C calls for I/O functionality with full ANSI/ISO standard I/O support with host machine console and file I/O for debugging sessions (Host I/O) through the JTAG port or HSST in addition to such standard C calls such as memory functions malloc() and free().

A minimal "thin" printf via "console_write" and "fflush_console" is provided in addition to standard I/O.

See the *MSL C Reference* manual (Metrowerks Standard Library).

Library Configurations

There are Large Data Model and Small Data Model versions of all libraries. (Small Program Model default is off for all library and Stationery targets.)

Metrowerks Standard Library (MSL) provides standard C library support.

The Runtime libraries provide the target-specific low-level functions below the high-level MSL functions. There are two types of Runtime libraries:

- JTAG-based Host I/O

- HSST-based Host I/O.

  For each project requiring standard C library support, a matched pair of MSL and Runtime libraries are required (SDM or LDM pairs).

  The HSST library is added to HSST client-to-client DSP56800E targets. For more information see "High-Speed Simultaneous Transfer" on page 251.

NOTE DSP56800E stationery creates new projects with LDM and SDM targets and the appropriate libraries.

Below is a list of the DSP56800E libraries:

- Metrowerks Standard Libraries (MSL)
  - `MSL C 56800E.lib`

    Standard C library support for Small Data Model.
  - `MSL C 56800E lmm.lib`

    Standard C library support for Large Data Model.
- Runtime Libraries
  - `runtime 56800E.lib`

    Low-level functions for MSL support for Small Data Model with Host I/O via JTAG port.
  - `runtime 56800E lmm.lib`

    Low-level functions for MSL support for Large Data Model with Host I/O via JTAG port.
  - `runtime_hsst_56800E.lib`

    Low-level functions for MSL support for Small Data Model with Host I/O via HSST.
  - `runtime_hsst_56800E_lmm.lib`

    Low-level functions for MSL support for Large Data Model with Host I/O via HSST.
- HSST Libraries

  There are debug and release targets for SDM and LDM. The release targets have maximum optimization settings and debug info turned off. For more information see "High-Speed Simultaneous Transfer" on page 251.

- `hsst_56800E.lib`

  DSP 56800E HSST client functions for Small Data Model.

- `hsst_56800E_lmm.lib`

  DSP56800E HSST client functions for Large Data Model.

Host File Location

Files are created with `fopen` on the host machine as shown in Table 10.1.

Table 10.1 Host File Creation Location

fopen filename parameter	host creation location
filename with no path	target project file folder
full path	location of full path

Allocating Stacks and Heaps for the DSP56800E

Stationery linker command files (LCF) define heap, stack, and bss locations. LCFs are specific to each target board. When you use M56800E stationery to create a new project, CodeWarrior automatically adds the LCF to the new project.

See "ELF Linker and Command Language," for general LCF information. See each specific target LCF in Stationery for specific LCF information.

See Table 10.2 for the variables defined in each Stationery LCF.

Table 10.2 LCF Variables and Address

Variables	Address
_stack_addr	the start address of the stack
_heap_size	the size of the heap
_heap_addr	the start address of the heap
_heap_end	the end address of the heap

Variables	Address
_bss_start	start address of memory reserved for uninitialized variables
_bss_end	end address of bss

To change the locations of these default values, modify the linker command file in your DSP56800E project.

NOTE Ensure that the stack and heap memories reside in data memory.

Definitions

Stack

The stack is a last-in-first-out (LIFO) data structure. Items are pushed on the stack and popped off the stack. The most recently added item is on top of the stack. Previously added items are under the top, the oldest item at the bottom. The "top" of the stack may be in low memory or high memory, depending on stack design and use. M56800E uses a 16-bit-wide stack.

Heap

Heap is an area of memory reserved for temporary dynamic memory allocation and access. MSL uses this space to provide heap operations such as malloc. M56800E does not have an operating system (OS), but MSL effectively synthesizes some OS services such as heap operations.

BSS

BSS is the memory space reserved for uninitialized data. The compiler will put all uninitialized data here. If "Zero initialized globals live in data instead of BSS" on page 63 is checked then the globals that are initialized to zero reside in the `.data` section instead of the `.bss` section. The stationery init code zeroes this area at startup. See the M56852 init (startup) code in this chapter for general information and the stationery init code files for specific target implementation details.

NOTE Instead of accessing the original Stationery files themselves (in the Stationery folder), create a new project using Stationery which will make copies of the specific target board files such as the LCF.

Runtime Initialization

The default init function is the bootstrap or glue code that sets up the DSP56800E environment before your code executes. This function is in the init file for each board-specific stationery project. The routines defined in the init file performs other tasks such as clearing the hardware stack, creating an interrupt table, and retrieving the stack start and exception handler addresses.

The final task performed by the init function is to call the `main()` function.

The starting point for a program is set in the **Entry Point** field in the **M56800E Linker** settings panel.

The project for the DSP56800E runtime is:

```
CodeWarrior\M56800E Support\runtime_56800E\projects\Runtime 56800E.mcp
```

Table 10.3 Library Names and Locations

Library Name	Location
Large Memory Model `Runtime 56800E lmm.lib`	`CodeWarrior\M56800E Support\runtime_56800E\lib`
Small Memory Model `Runtime 56800E.Lib`	`CodeWarrior\M56800E Support\runtime_56800E\lib`

When creating a project from R1.1 or later Stationery, the associated init code is specific to the DSP56800E board. See the startup folder in the new project folder for the init code.

Listing 10.1 Sample Initialization File (DSP56852EVM)

```
#

; ---------------------------------------------------------
;
;       56852_init.asm
;       Metrowerks, a Motorola Company
;       sample
        description:  main entry point to C code.
;                 setup runtime for C and call main
;
; ---------------------------------------------------------


;===============================
; OMR mode bits
;===============================
NL_MODE                 EQU             $8000
CM_MODE                 EQU             $0100
XP_MODE                 EQU             $0080
R_MODE                  EQU             $0020
SA_MODE                 EQU             $0010


        section rtlib

        XREF F_stack_addr
        org p:


        GLOBAL Finit_M56852_

        SUBROUTINE "Finit_M56852_",Finit_M56852_,Finit_M56852END-Finit_M56852_

Finit_M56852_:

;
; setup the OMr with the values required by C
;
        bfset   #NL_MODE,omr          ; ensure NL=1  (enables nsted DO loops)
        nop
        nop
        bfclr   #(CM_MODE|XP_MODE|R_MODE|SA_MODE),omr   ; ensure CM=0
(optional for C)
                  ; ensure XP=0 to enable harvard architecture
                  ; ensure R=0  (required for C)
                  ; ensure SA=0 (required for C)
```

```
; Setup the m01 register for linear addressing
        move.w  #-1,x0
        moveu.w x0,m01              ; Set the m register to linear addressing

        moveu.w hws,la              ; Clear the hardware stack
        moveu.w hws,la
        nop
        nop


CALLMAIN:                           ; Initialize compiler environment


;Initialize the Stack
        move.l #>>F_Lstack_addr,r0
        bftsth #$0001,r0
        bcc noinc
        adda #1,r0
noinc:
        tfra    r0,sp                ; set stack pointer too
        move.w  #0,r1
        nop
        move.w  r1,x:(sp)
        adda    #1,sp

  jsr                  F__init_sections

; Call main()
        move.w  #0,y0               ; Pass parameters to main()
        move.w  #0,R2
        move.w  #0,R3

        jsr            Fmain        ; Call the Users program
;
;  The fflush calls where removed because they added code
;  growth in cases where the user is not using any debugger IO.
;  Users should now make these calls at the end of main if they use debugger IO
;
;       move.w  #0,r2
;       jsr     Ffflush             ; Flush File IO
;       jsr     Ffflush_console     ; Flush Console IO

;       end of program; halt CPU
        debughlt
        rts
Finit_M56852END:

       endsec
```

EOnCE Library

The EOnCE (Enhanced On Chip Emulator) library provides functions, which allows your program to control the EOnCE. The library lets you set and clear triggers for breakpoints, watchpoints, program traces, and counters. With several option enumerations, the library greatly simplifies using the EOnCE from within the core, and thus eliminates the need for a DSP56800E User Manual. The library and the debugger are coordinated so that the debugger does not overwrite a trigger set by the library, and vice versa.

To use the EOnCE library, you must include it in your project. The name of the file is `eonce 56800E lmm.lib` and it is located at:

```
CodeWarrior\M56800ESupport\eonce\lib
```

The **Large Data Model** option must be enabled in the **M56800E Processor** preference panel. Any source file that contains code that calls any of the EOnCE Library functions must `#include eonceLib.h`. This header file is located at:

```
CodeWarrior\M56800E Support\eonce\include
```

The library functions are listed below:

- _eonce_Initialize
- _eonce_SetTrigger
- _eonce_SetCounterTrigger
- _eonce_ClearTrigger
- _eonce_GetCounters
- _eonce_GetCounterStatus
- _eonce_SetupTraceBuffer
- _eonce_GetTraceBuffer
- _eonce_ClearTraceBuffer
- _eonce_StartTraceBuffer
- _eonce_HaltTraceBuffer
- _eonce_EnableDEBUGEV
- _eonce_EnableLimitTrigger

The sub-section "Definitions" on page 334 defines:

- Return Codes
- Normal Trigger Modes
- Counter Trigger Modes
- Data Selection Modes
- Counter Function Modes
- Normal Unit Action Options
- Counter Unit Action Options
- Accumulating Trigger Options
- Miscellaneous Trigger Options
- Trace Buffer Capture Options
- Trace Buffer Full Options
- Miscellaneous Trace Buffer Option

_eonce_Initialize

Initializes the library by setting the necessary variables.

Prototype

```
void _eonce_Initialize( unsigned long baseAddr, unsigned int units )
```

Parameters

`baseAddr  unsigned long`

This parameter specifies the location in X: memory where the EOnCE registers are located.

`units  unsigned int`

This parameter specifies the number of EOnCE breakpoint units available.

Remarks

This function must be called before any other library function is called. Its parameters are dependent on the processor being used. Instead of calling this function directly, one of the defined macros can be called in its place. These include **_eonce_Initialize56838E()**, **_eonce_Initialize56852E()**, and **_eonce_Initialize56858E()**. These macros call **_eonce_Initialize** with the correct parameters for the 56838, 56852, and 56858, respectively.

Returns

Nothing.

_eonce_SetTrigger

Sets a trigger condition used to halt the processor, cause an interrupt, or start and stop the trace buffer. This function does not set triggers for special counting functions.

Prototype

```
int _eonce_SetTrigger( unsigned int unit, unsigned
    long options, unsigned long value1, unsigned
    long value2, unsigned long mask, unsigned int
    counter )
```

Parameters

`unit` `unsigned int`

This parameter specifies which breakpoint unit to use.

`options` `unsigned long`

This parameter describes the behavior of the trigger. For more information on the identifiers for this parameter, please see the sub-section "Definitions" on page 313.

`value1` `unsigned long`

This parameter specifies the address or data value to compare as defined by the options parameter.

`value2` `unsigned long`

This parameter specifies the address or data value to compare as defined by the options parameter.

`mask` `unsigned long`

This parameter specifies which bits of value2 to compare.

`counter` `unsigned int`

This parameter specifies the number of successful comparison matches to count before completing trigger sequence as defined by the options parameter

Remarks

This function sets all triggers, except those used to define the special counting function behavior. Carefully read the list of defined identifiers that can be OR'ed into the options parameter.

Returns

Error code as defined in the sub-section "Definitions" on page 334.

_eonce_SetCounterTrigger

Sets a trigger condition used for special counting functions.

Prototype

```
int _eonce_SetCounterTrigger( unsigned int unit,
    unsigned long options, unsigned long value1, un-
    signed long value2, unsigned long mask, unsigned
    int counter, unsigned long counter2 )
```

Parameters

`unit` `unsigned int`

This parameter specifies which breakpoint unit to use.

`options` `unsigned long`

This parameter describes the behavior of the trigger. For more information on the identifiers for this parameter, please see the sub-section "Definitions" on page 313.

`value1` `unsigned long`

This parameter specifies the address or data value to compare as defined by the options parameter.

`value2` `unsigned long`

This parameter specifies the address or data value to compare as defined by the options parameter.

`mask` `unsigned long`

This parameter specifies which bit of value2 to compare.

`counter` `unsigned int`

This parameter specifies the value used to pre-load the counter, which proceeds backward when `EXTEND_COUNTER` is OR'ed into the options parameter. `counter` contains the least significant 16-bits.

`counter2` `unsigned long`

This parameter specifies the value used to pre-load the counter, which proceeds backward. When `EXTEND_COUNTER` is OR'ed into the options parameter. `counter2` contains the most significant 24-bits. However, when `EXTEND_COUNTER` is not OR'ed

`counter2` should be set to `0`.

Remarks — This function is used to set special counting function triggers. The special counting options are defined in the sub-section "Definitions" on page 334. Carefully read the list of defined identifiers that can be OR'ed into the options parameter.

Returns — Error code as defined in the sub-section "Definitions" on page 334.

_eonce_ClearTrigger

Clears a previously set trigger.

Prototype `int _eonce_ClearTrigger( unsigned int unit )`

Parameters `unit  unsigned int`

This parameter specifies which breakpoint unit to use.

Remarks This function clears a trigger set with the **_eonce_SetTrigger** or **_eonce_SetCounterTrigger** functions.

Returns Error code as defined in the sub-section "Definitions" on page 334.

_eonce_GetCounters

Retrieves the values in the two counter registers.

Prototype

```
int _eonce_GetCounters( unsigned int unit, unsigned
    int *counter, unsigned long *counter2 )
```

Parameters

`unit  unsigned int`

This parameter specifies which breakpoint unit to use.

`counter  unsigned int *`

This parameter holds the value of the counter, or the least significant 16-bits, if the counter has been extended to 40-bits.

`counter2  unsigned long *`

This parameter holds the most significant 24-bits if the counter has been extended to 40-bits. This parameter must be a valid pointer even if the counter has not been extended.

Remarks

This function retrieves the value of the counter of the specified breakpoint unit. This function is most useful when using the special counting function of the breakpoint, but can also be used to retrieve the trigger occurrence counter.

Returns

Error code as defined in the sub-section "Definitions" on page 334.

_eonce_GetCounterStatus

Retrieves the status of the breakpoint counter.

Prototype

```
int _eonce_GetCounters( char *counterIsZero, char
    *counterIsStopped )
```

Parameters

`counterIsZero  char *`

This parameter returns a 1 if the breakpoint counter has reached zero.

`counterIsStopped  char *`

This parameter returns a 1 if the breakpoint counter has been stopped by a Counter Stop Trigger.

Remarks

This function returns the state of the breakpoint counter when using the special counting function.

Returns

Error code as defined in the sub-section "Definitions" on page 334.

_eonce_SetupTraceBuffer

Configures the behavior of the trace buffer.

Prototype `int _eonce_SetupTraceBuffer( unsigned int options )`

Parameters `options  unsigned int`

This parameter describes the behavior of the trace buffer. Please see the section Definitions for more information on the identifiers for this parameter.

Remarks This function sets the behavior of the trace buffer. Triggers can also be set to start and stop trace buffer capture using the **_eonce_SetTrigger** function.

Returns Error code as defined in the sub-section "Definitions" on page 334.

_eonce_GetTraceBuffer

Retrieves the contents of the trace buffer.

Prototype

```
int _eonce_GetTraceBuffer( unsigned int *count, unsigned long *buffer )
```

Parameters

`count` `unsigned int *`

This parameter passes in the size of the buffer; if 0 is passed in, the contents of the trace buffer are not retrieved, instead the number of entries in the trace buffer are returned in count.

`buffer` `unsigned long *`

This parameter points to an array in which the contents of the trace buffer are returned starting with the oldest entry.

Remarks

This function retrieves the addresses contained in the trace buffer. The addresses represent the program execution point when certain change-of-flow events occur. The trace buffer behavior, including capture events, can be configured using **_eonce_SetupTraceBuffer**.

Returns

Error code as defined in the sub-section "Definitions" on page 334.

_eonce_ClearTraceBuffer

Clears the contents of the trace buffer.

Prototype `int _eonce_ClearTraceBuffer( )`

Parameters There are no parameters for this function.

Remarks This function clears the trace buffer and is useful when you want a fresh set of data. It is necessary to resume capturing when the trace buffer is full and configured to stop capturing.

Returns Error code as defined in the sub-section "Definitions" on page 334.

_eonce_StartTraceBuffer

Resumes trace buffer capturing.

Prototype `int _eonce_StartTraceBuffer( )`

Parameters There are no parameters for this function.

Remarks This function causes the trace buffer to immediately start capturing.

Returns Error code as defined in the sub-section "Definitions" on page 334.

_eonce_HaltTraceBuffer

Halts trace buffer capturing.

Prototype `int _eonce_HaltTraceBuffer( )`

Parameters There are no parameters for this function.

Remarks This function causes the trace buffer to immediately stop capturing.

Returns Error code as defined in the sub-section "Definitions" on page 334.

_eonce_EnableDEBUGEV

Allows or disallows a DEBUGEV instruction to cause a core event in breakpoint unit 0.

Prototype `int _eonce_EnableDEBUGEV( char enable )`

Parameters `enable  char`

This parameter can specify a non-zero or zero value.

If it is a non-zero value, then the DEBUGEV instruction causes a core event.

If it is a zero value, then the action is disallowed.

Remarks This function configures the behavior for the DEBUGEV instructions. For a core event to occur, breakpoint unit 0 must be activated by setting a trigger using the **_eonce_SetTrigger** or **_eonce_SetCounterTrigger** functions.

Returns Error code as defined in the sub-section "Definitions" on page 334.

_eonce_EnableLimitTrigger

Allows or disallows a limit trigger to cause a core event in breakpoint unit 0.

Prototype

```
int _eonce_EnableLimitTrigger( char enable )
```

Parameters

`enable` `char`

This parameter can specify a non-zero or zero value.

If it is a non-zero value, it allows a limit trigger to cause a core event.

If it is a zero value, it disallows the action.

Remarks

This function configures the behavior for overflow and saturation conditions in the processor core. For a core event to occur, breakpoint unit 0 must be activated by setting a trigger using the **_eonce_SetTrigger** or **_eonce_SetCounterTrigger** functions.

Returns

Error code as defined in the sub-section "Definitions" on page 334.

Definitions

This sub-section defines:

- Return Codes
- Normal Trigger Modes
- Counter Trigger Modes
- Data Selection Modes
- Counter Function Modes
- Normal Unit Action Options
- Counter Unit Action Options
- Accumulating Trigger Options
- Miscellaneous Trigger Options
- Trace Buffer Capture Options
- Trace Buffer Full Options
- Miscellaneous Trace Buffer Option

Return Codes

Every function except **_eonce_Initialize** returns one of the error codes in Table 10.4.

Table 10.4 Error Codes

Error Code	Description
EONCE_ERR_NONE	No error.
EONCE_ERR_NOT_INITIALIZED	The **_eonce_Initialize** function has not been called before the current function.
EONCE_ERR_UNIT_OUT_OF_RANGE	The unit parameter is greater than or equal to the number of units specified in **_eonce_Initialize.**
EONCE_ERR_LOCKED_OUT	The core cannot access the EOnCE registers because the debugger has locked out the core. This occurs when a trigger has been set using the EOnCE GUI panels or through an IDE breakpoint or watchpoint.

Normal Trigger Modes

One of the defined identifiers listed in Listing 10.2 must be OR'ed into the options parameter of the **_eonce_SetTrigger** function. A key for the defined identifiers listed in Listing 10.2 is given in Table 10.5.

Listing 10.2 Normal Trigger Modes

```
B1PA_N
B1PR_N
B1PW_N
B2PF_N
B1XA_OR_B2PF_N
B1XA_N_OR_B2PF
B1PF_OR_B2PF_N
B1PA_OR_B2PF_N
B1PA_N_OR_B2PF
B1PF_OR_N_B2PF
B1PA_OR_N_B2PF
B1XR_AND_N_B2DR
B1XW_AND_N_B2DW
B1XA_AND_N_B2DRW
B1PF_N_THEN_B2PF
B2PF_THEN_B1PF_N
B1PA_N_THEN_B2PF
B1PA_THEN_B2PF_N
B2PF_N_THEN_B1PA
B2PF_THEN_B1PA_N
B1XA_N_THEN_B2PF
B1XA_THEN_B2PF_N
B2PF_N_THEN_B1XA
B2PF_THEN_B1XA_N
B1XW_N_THEN_B2PF
B1XW_THEN_B2PF_N
B2PF_N_THEN_B1XW
B2PF_THEN_B1XW_N
B1XR_N_THEN_B2PF
B1XR_THEN_B2PF_N
B2PF_N_THEN_B1XR
B2PF_THEN_B1XR_N
B1PF_STB_B2PF_HTB
```

```
B1PA_STB_B2PF_HTB
B2PF_STB_B1PA_HTB
```

Table 10.5 Defined Identifier Key for Normal Trigger Modes

Identifier Fragments	Description
B1	breakpoint 1; value set in value1
B2	breakpoint 2; value set in value2
P	p-memory address; this is followed by a type of access
X	x-memory address; this is followed by a type of access
D	value being read from or written to x-memory
A	memory access
R	memory read
W	memory write
F	memory fetch; only follows a P
OR	links two sub-triggers by a logical or
AND	links two sub-triggers by a logical and
THEN	creates a sequence; first sub-trigger must occur, then second sub-trigger must occur to complete the trigger
N	the sub-trigger it follows must occur N times as set in the count parameter; if N follows an operation, then the combination of the sub-triggers must occur N times; (count - 1) will be written to the BCNTR register
STB	sub-trigger starts the trace buffer
HTB	sub-trigger halts the trace buffer

Counter Trigger Modes

The following triggers generate a Counter Stop Trigger. The exceptions are the modes that generate both start and stop triggers.

The defined identifiers listed in Listing 10.3 must be OR'ed into the options parameter of the **_eonce_SetCounterTrigger** function. A key for the defined identifiers listed in Listing 10.3 is given in Table 10.6

Listing 10.3 Counter Trigger Modes

```
B1PA
B1PR
B1PW
B2PF
B1XA_OR_B2PF
B1PF_OR_B2PF
B1PA_OR_B2PF
B1XR_AND_B2DR
B1XW_AND_B2DW
B1XA_AND_B2DRW
B1PF_THEN_B2PF
B1PA_THEN_B2PF
B2PF_THEN_B1PA
B1XA_THEN_B2PF
B2PF_THEN_B1XA
B1XW_THEN_B2PF
B2PF_THEN_B1XW
B1XR_THEN_B2PF
B2PF_THEN_B1XR
B1PF_SC_B2PF_HC
B1PA_SC_B2PF_HC
B2PF_SC_B1PA_HC
```

Table 10.6 Defined Identifier Key for Counter Trigger Modes

Identifier Fragments	Description
B1	breakpoint 1; value set in value1
B2	breakpoint 2; value set in value2
P	p-memory address; this is followed by a type of access
X	x-memory address; this is followed by a type of access.
D	value being read from or written to x-memory
A	memory access
R	memory read
W	memory write
F	memory fetch; only follows a P
OR	links two sub-triggers by a logical or
AND	links two sub-triggers by a logical and
THEN	creates a sequence; first sub-trigger must occur, then second sub-trigger must occur to complete the trigger
SC	sub-trigger starts the counter
HC	sub-trigger halts the counter

Data Selection Modes

If the trigger mode being set includes a data value compare (contains B2D from the list Normal Trigger Modes or Counter Trigger Modes), then one of the defined identifiers in Table 10.7 must be OR'ed into the options parameter of the **_eonce_SetTrigger** or **_eonce_SetCounterTrigger** function. If not, then do not OR in any of these identifiers.

Table 10.7 Data Selection Modes

Defined Identifiers	Description
B2D_BYTE	makes a comparison when the data being moved is of byte-length
B2D_WORD	makes a comparison when the data being moved is of word-length
B2D_LONG	makes a comparison when the data being moved is of long-length

Counter Function Modes

One of the defined identifiers in Table 10.8 must be OR'ed into the options parameter of the **_eonce_SetCounterTrigger** function.

Table 10.8 Counter Function Modes

Defined Identifiers	Description
PCLK_CLOCK_CYCLES	count pclk cycles
CLK_CLOCK_CYCLES	count clk cycles
INSTRUCTIONS_EXECUTED	count instructions executed
TRACE_BUFFER_WRITES	count writes to the trace buffer
COUNTER_START_TRIGGERS	count Counter Start Triggers

Normal Unit Action Options

This list of options describes the action taken when a non-counter trigger is generated. One of the defined identifiers in Table 10.9 must be OR'ed into the options parameter of the **_eonce_SetTrigger** function.

Table 10.9 Normal Unit Actions Options Mode

Defined Identifier	Description
UNIT_ACTION	enters debug mode is unit 0, else passes signal on to next unit
INTERRUPT_CORE	interrupts to vector set for this unit
HALT_TRACE_BUFFER	trace buffer capture is halted
START_TRACE_BUFFER	trace buffer capture is started

Counter Unit Action Options

This list of options describes the action taken when a counter trigger is generated. One of the defined identifiers in Table 10.10 must be OR'ed into the options parameter of the **_eonce_SetCounterTrigger** function. Identifiers that include ZERO_BEFORE_TRIGGER only perform the action when the counter counts down to zero before the Counter Stop Trigger occurs. Identifiers that include TRIGGER_BEFORE_ZERO only perform the action when the Counter Stop Trigger occurs before the counter counts down to zero.

Table 10.10 Counter Unit Actions Options Mode

Defined Identifier	Description
NO_ACTION	counter status bits still get set
UNIT_ACTION_ZERO_BEFORE_TRIGGER	enters debug mode is unit 0, else passes signal on to next unit
INTERRUPT_CORE_ZERO_BEFORE_TRIGGER	interrupts to vector set for this unit
UNIT_ACTION_TRIGGER_BEFORE_ZERO	enters debug mode is unit 0, else passes signal on to next unit
INTERRUPT_CORE_TRIGGER_BEFORE_ZERO	interrupts to vector set for this unit

Accumulating Trigger Options

One of the defined identifiers in Table 10.11 must be OR'ed into the options parameter of the **_eonce_SetTrigger** function when breakpoint unit 0 is being configured.

Table 10.11 Accumulating Trigger Options Mode with Breakpoint Unit 0

Defined Identifier	Description
PREV_UNIT_OR_THIS_TRIGGER_OR_CORE_EVENT	a trigger is generated if the previous breakpoint unit passes in a trigger signal or this breakpoint unit creates a trigger signal or if a core event occurs
PREV_UNIT_THEN_THIS_TRIGGER_OR_CORE_EVENT	a trigger is generated if the previous breakpoint unit passes in a trigger signal followed by either this breakpoint unit creating a trigger signal or a core event occurring
THIS_TRIGGER_THEN_CORE_EVENT	a trigger is generated if this breakpoint unit creates a trigger signal followed by a core event occurring
PREV_UNIT_THEN_THIS_TRIGGER_THEN_CORE_EVENT	a trigger is generated if the previous breakpoint unit passes in a trigger signal followed by this breakpoint unit creating a trigger signal followed by a core event occurring

One of the defined identifiers in Table 10.12 must be OR'ed into the options parameter of the **_eonce_SetTrigger** function when a breakpoint unit other than unit 0 is being configured.

Table 10.12 Accumulating Trigger Options Mode with Breakpoint Unit other than 0

Defined Identifier	Description
PREV_UNIT_OR_THIS_TRIGGER	a trigger is generated if the previous breakpoint unit passes in a trigger signal or this breakpoint unit creates a trigger signal
PREV_UNIT_THEN_THIS_TRIGGER	a trigger is generated if the previous breakpoint unit passes in a trigger signal followed by this breakpoint unit creating a trigger signal

Miscellaneous Trigger Options

The defined identifiers in Table 10.13 are optional.

Table 10.13 Miscellaneous Trigger Options

Defined Identifier	Description
INVERT_B2_COMPARE	the signal from breakpoint 2 is inverted before entering the combination logic; this can be OR'ed into the options parameter of the **_eonce_SetTrigger** or **_eonce_SetCounterTrigger** function
EXTEND_COUNTER	the counter, when using the special counting function, is extended to 40-bits by using the OSCNTR as the most significant 24-bits; this can be OR'ed into the options parameter of the **_eonce_SetCounterTrigger** function when configuring breakpoint unit 0; WARNING: It is not recommended that this option be used if the processor will enter debug mode (breakpoint, console or file I/O) before the counter is read, because the OSCNTR is needed for stepping and would corrupt the counter

Trace Buffer Capture Options

The options in Table 10.14 determine which kind of changes-of-flow will be captured. OR in as many of the following defined identifiers into the options parameter of the **_eonce_SetupTraceBuffer** function.

Table 10.14 Trace Buffer Capture Options

Defined Identifier	Description
CAPTURE_CHANGE_OF_FLOW_NOT_TAKEN	saves target addresses of conditional branches and jumps that are not taken to the trace buffer
CAPTURE_CHANGE_OF_FLOW_INTERRUPT	saves addresses of interrupt vector fetches and target addresses of RTI instructions to the trace buffer
CAPTURE_CHANGE_OF_FLOW_SUBROUTINE	saves the target addresses of JSR, BSR, and RTS instructions to the trace buffer
CAPTURE_CHANGE_OF_FLOW_0	saves the target addresses of the following taken instructions to the trace buffer: BCC forward branch BRSET forward branch BRCLR forward branch JCC forward and backward branches
CAPTURE_CHANGE_OF_FLOW_1	saves the target addresses of the following taken instructions to the trace buffer: BCC backward branch BRSET backward branch BRCLR backward branch

Trace Buffer Full Options

The options in Table 10.15 describe what action to take when the trace buffer is full. One of the following defined identifiers must be OR'ed into the options parameter of the **_eonce_SetupTraceBuffer** function.

Table 10.15 Trace Buffer Full Options

Defined Identifier	Description
TB_FULL_NO_ACTION	capture continues, overwriting previous entries
TB_FULL_HALT_CAPTURE	capture is halted
TB_FULL_DEBUG	processor enters debug mode
TB_FULL_INTERRUPT	processor interrupts to vector specified as Trace Buffer Interrupt

Miscellaneous Trace Buffer Option

The TRACE_BUFFER_HALTED option may be OR'ed into the options parameter of the **_eonce_SetupTraceBuffer** function. This option puts the trace buffer in a halted state when leaving **_eonce_SetupTraceBuffer** function. This is most useful when setting a trigger, by calling **_eonce_SetTrigger**, to start the trace buffer when a specific condition is met.

Index

Symbols

__mod_access 160
__mod_error 162
__mod_getint16 161
__mod_init 159
__mod_initint16 160
__mod_setint16 162
__mod_start 160
__mod_stop 161
__mod_update 161
_eonce_ClearTraceBuffer 329
_eonce_ClearTrigger 324
_eonce_EnableDEBUGEV 332
_eonce_EnableLimitTrigger 333
_eonce_GetCounters 325
_eonce_GetCounterStatus 326
_eonce_GetTraceBuffer 328
_eonce_HaltTraceBuffer 331
_eonce_Initialize 320
_eonce_SetCounterTrigger 322
_eonce_SetTrigger 321
_eonce_SetupTraceBuffer 327
_eonce_StartTraceBuffer 330

A

abs_s 124
Access Paths panel 36, 40
access permission flags 287
accessing data object
    global pointer variable 94
accessing data objects
    examples 93
    global integer 93
Accumulating Trigger Options 340
add 126
Add Files command 27
add_hfm_unit 77, 247
adding assembly language 113
addr 284
AFTER command 287
align 284
alignall 285
alignment, stack 87
alignsp, pragma interrupt 99
Allocating Memory and Heaps for DSP56800E 312
`asm` keyword 112
asmoutput 97
assembly language 111
    statements, adding 113

B

bootstrap code 315
breakpoints 197
Bring Up To Date command 32
Build Extras panel 40
Build System 33
build target, defined 16

C

C language
    general notes, for DSP56800E 79
C/C++ Language panel 45, 46
C/C++ Warnings panel 51
'called' option in #pragma interrupt 102
called, pragma interrupt 100
calling assembly functions from C code 115
Calling Conventions and Stack Frames for DSP56800E 82
changing PCI connection, command converter server 174
Changing Target Settings 37
`char` size 80
character data addressing, additional information 90
check_c_src_pipeline 98
check_inline_asm_pipeline 98
code and data storage for DSP56800E 89
code, deadstripping unused 109
code, editing 28
CodeWarrior
    compiler architecture 33, 34
    debugging for DSP56800E 169
CodeWarrior IDE
    available tools 16
    comparison to command line 31

development process 31
installing 20
CodeWarrior IDE and Its Documentation 15
CodeWarrior IDE Target Settings Panels 40
Command Converter Server 174
changing PCI connection 174
changing the parallel port 172
local connection 71
target settings 171
command converter server
icon 172
menu 172
Command Converter Server window 173
command line and CodeWarrior IDE compared 31
command-line debugging
clashing Tcl commands, resolving 208
commands
alias 213
break 214
bringtofront 215
cd 216
change 217
close 219
cls 219
config 219
copy 221
debug 222
dir 222
disassemble 223
display 224
evaluate 226
exit 227
go 227
help 228
history 228
hsst_attach_listener 229
hsst_block_mode 229
hsst_close 230
hsst_detach_listener 230
hsst_log 230
hsst_noblock_mode 231
hsst_open 231
hsst_read 231
hsst_write 232
input 232
kill 233
load 234
log 235
ls 222
next 236
output 236
pwd 237
radix 237
restart 239
run 239
save 240
step 241
stop 242
switchtarget 243
system 243
view 244
wait 244
watchpoint 245
introduction 208
tasks
clear a command from the command line 212
copy text from the command-line debugging window 213
enter a single command 210
enter multiple commands 210
open a command-line debugging window 210
paste text into the command-line debugging window 213
repeat a command 211
review previously entered commands 211
scroll text in the command-line debugging window 212
switch between insert and overwrite mode 212
view debugging command hints 211
Tcl script files, executing 209
Tcl start-up script, using 209
Tcl support 208
Command-line tools
arguments 296
response file 294
sample build script 294
usage 293
Command-line tools, for DSP56800E 293
commands
Add Files 27
Bring Up To Date 32
Compile 32
Enable Debugger 34
Flash Memory 246
Make 34

Preprocess 34
Compile command 32
compiler
architecture 33, 34
intermediate representation (IR) 33
plug-in modules, explained 33
support for inline assembly 111
compiling 32
See also IDE User Guide
comr, pragma interrupt 99
Connect menu item 245
Connection list menu 70
Console 310
Console and File I/O 310
Contents of Trace Buffer panel 189
Counter Function Modes 339
Counter Trigger Modes 336
Counter Unit Action Options 340
Create a New Project dialog box 25
Creating a Project 24
creating labels for DSP56800E Assembly 114
Custom Keywords settings panel 40

D

data alignment requirements, for DSP56800E 87
Data Selection Modes 338
deadstripping unused code and data 109
deadstripping, prevention 271, 275
Debugger Settings panel 40
debugger, setting preferences 169, 194
debugging 34
Command Converter Server 170
ELF files 296
Flash Memory 246
projects 195
Register Details Window 201
Target panel 169
without symbolics 245
See also IDE User Guide
debugging a remote target board 177
default, pragma interrupt 100
define_section 106
defining an inline assembly function 113
definition
BSS 313
heap 313
stack 313
Disable Deadstripping checkbox 68
div_ls 135
div_ls4q 136
div_s 134
div_s4q 135
`double` size, floating-point type 81
DSP56800E
calling conventions and stack frames 82
code and data storage 89
floating-point formats 81
integer formats 80
linker 109
`pragma` directives 97
Simulator 193
stack frame 86

E

editing
code 32
project contents 27
source files 28
See also IDE User Guide
editor, of IDE 28
ELF Disassembler settings panel 58
Show Addresses and Object Code checkbox 59
Show Code Modules checkbox 59
Show Comments checkbox 60
Show Data Modules checkbox 60
Show Debug Info checkbox 60
Show Headers checkbox 58
Show Relocations checkbox 59
Show Source Code checkbox 59
Show Symbol and String Tables checkbox 58
Use Extended Mnemonics checkbox 59
Verbose Info checkbox 59
Enable Debugger command 34
EOnCE Debugger Features 184
set hardware breakpoints 185
set trigger panel 190
special counters 186
trace buffer 187
EOnCE Definitions
Accumulating Trigger Options 340
Counter Function Modes 339
Counter Trigger Modes 336
Counter Unit Action Options 340
Data Selection Modes 338

Miscellaneous Trace Buffer Options 344
Miscellaneous Trigger Options 342
Normal Trigger Modes 335
Normal Unit Action Options 339
Return Codes 334, 335
Trace Buffer Capture Options 342
Trace Buffer Full Options 343
EOnCE Library 318
_eonce_ClearTraceBuffer 329
_eonce_ClearTrigger 324
_eonce_EnableDEBUGEV 332
_eonce_EnableLimitTrigger 333
_eonce_GetCounters 325
_eonce_GetCounterStatus 326
_eonce_GetTraceBuffer 328
_eonce_HaltTraceBuffer 331
_eonce_Initialize 320
_eonce_SetCounterTrigger 322
_eonce_SetTrigger 321
_eonce_SetupTraceBuffer 327
_eonce_StartTraceBuffer 330
EOnCE Special Counter panel 186
Error 166
Exporting and importing panel options to XML Files 38
extended data addressing
examples 92
external library compatibility 94
extract_h 132
extract_l 132

F

ffs_l 143
ffs_s 142
File Mappings panel 40
fileName 288
fill memory 180
fill memory dialog box 180
Flash Memory
commands 246
debugging 246
Flash security features 248
`float` size, floating-point type 81
floating-point formats, for DSP56800E 81
Force Active Symbols text box 70
force_active 271, 275, 285
fractional arithmetic 122
equation for converting 122

G

general notes on C, for DSP56800E 79
general purpose registers 199
Generate ELF Symbol Table checkbox 68
Generate Link Map checkbox 66
Generate Listing File checkbox 55
Generate S-Record File checkbox 68
Generate Symbolic Info checkbox 65
GLOBAL directive, assembly function definitions 116
global integer 93
Global Optimizations settings panel 40
global pointer variable 94
global variables, linker command file 275
group keyword 278, 286

H

Hardware DO Loops 62
heap size 281
High-Speed Simultaneous Transfer (HSST)
host-side client interface
hsst_attach_listener 256
hsst_block_mode 255
hsst_close 252
hsst_detach_listener 257
hsst_noblock_mode 256
hsst_open 252
hsst_read 253
hsst_set_log_dir 257
hsst_size 255
hsst_write 254
introduction 251
target library interface
HSST_close 260
HSST_flush 263
HSST_open 260
HSST_raw_read 264
HSST_raw_write 265
HSST_read 263
HSST_set_log_dir 266
HSST_setvbuf 260
HSST_size 264
HSST_write 262
introduction 259
host, defined 15

HSST example
Host Program 258
Target Program 266
HSST Host Program example 258
HSST Target Program example 266
hsst_attach_listener 256
hsst_attach_listener debugging command 229
hsst_block_mode 255
hsst_block_mode debugging command 229
HSST_close 260
hsst_close 252
hsst_close debugging command 230
hsst_detach_listener 257
hsst_detach_listener debugging command 230
HSST_flush 263
hsst_log debugging command 230
hsst_noblock_mode 256
hsst_noblock_mode debugging command 231
HSST_open 260
hsst_open 252
hsst_open debugging command 231
HSST_raw_read 264
HSST_raw_write 265
HSST_read 263
hsst_read 253
hsst_read debugging command 231
HSST_set_log_dir 266
hsst_set_log_dir 257
HSST_setvbuf 260
HSST_size 264
hsst_size 255
HSST_write 262
hsst_write 254
hsst_write debugging command 232

I

icon, command converter server 172
implied fractional value 122
include keyword 286
init function 315
inline assembly
defining functions 113
function-level 112
instructions 112
optimization 115
statement-level 112
syntax 112
installing, CodeWarrior IDE 20
`int` size, floating-point type 80
integer formats, for DSP56800E 80
integral types, in LCF 275
interrupt 98
arguments 99
intrinsic functions
absolute/negate 124
abs_s 124
L_abs 125
L_negate 125
negate 124
addition/subtraction 126
add 126
L_add 127
L_sub 128
sub 127
control 129
stop 129
turn_off_conv_rndg 130
turn_off_sat 130
turn_on_conv_rndg 130
turn_on_sat 131
wait 129
deposit/extract 132
extract_h 132
extract_l 132
L_deposit_h 133
L_deposit_l 133
division 134
div_ls 135
div_ls4q 136
div_s 134
div_s4q 135
for math support 123
modulo addressing 157
multiplication/MAC 137
L_mac 139
L_msu 140
L_mult 140
L_mult_ls 141
mac_r 137
msu_r 138
mult 138
mult_r 139
normalization 142
ffs__s 142
ffs_l 143

norm_l 144
norm_s 143
rounding 145
round 145
shifting 146
L_shl 153
L_shlftNs 153
L_shlfts 154
L_shr 155
L_shr_r 156
L_shrtNs 156
shl 147
shlftNs 148
shlfts 149
shr 150
shr_r 151
shrtNs 152

K

keep_section 271, 275, 286

L

L_abs 125
L_add 127
L_deposit_h 133
L_deposit_l 133
L_mac 139
L_msu 140
L_mult 140
L_mult_ls 141
L_negate 125
L_shl 153
L_shlftNs 153
L_shlfts 154
L_shr 155
L_shr_r 156
L_shrtNs 156
L_sub 128
labels, M56800E assembly 114
Large Data Model 91
large data model 62
large data model support 91
Launching and Operating the Debugger 194
LCF, *See* linker command files
LENGTH
autolength by specifying zero 288
value greater than zero 288
libraries
MSL for DSP56800E 309
support for DSP56800E 309
Library, EOnCE 318
license key 21
linear addressing 103
link order 110
linker
for DSP56800E 109
link order 110
See also linker command files
settings 65
linker command files 269
access permission flags 287
addr 284
AFTER command 287
align 284
alignall 285
alignment 273
arithmetic operations 274
command blocks 271
comments, using # character 274
deadstripping prevention 275
expressions, syntax 276
file selection 278
force_active 285
function selection 279
global variables 275
group keyword 278, 286
heap size 281
include keyword 286
integral types 275
keep_section 286
memory directive 287
memory segment 270
object 289
object keyword 279
ref_include 289
sections 289, 290
sections segment 272
sizeof 291
sizeofw 291
stack size 281
symbols 275
variables, type 275
writeb 291
writeh 292
writew 292

writing data to memory 282
Linker Issues for Motorola DSP56800E 109
Linker list box 42
linking 34
See also IDE User Guide
List Unused Objects checkbox 66
load/save memory 177
load/save memory dialog box 178
loading a .elf file without a project 207
`long double` size, floating-point type 81
`long` size, ordinal type 80

M

M56800E Assembler settings panel 55
Generate Listing File checkbox 55
M56800E Linker
Disable Deadstripping checkbox 68
Force Active Symbols text box 70
Generate ELF Symbol Table checkbox 68
Generate Symbolic Info checkbox 65
List Unused Objects checkbox 66
Show Transitive Closure checkbox 67
Store Full Path Names checkbox 66
M56800E Linker settings panel 65
Generate Link Map checkbox 66
Generate S-Record File checkbox 68
Max Record Length field 69
S-Record EOL Character list menu 69
Suppress Warning Messages checkbox 68
M56800E Processor settings panel 61
Hardware DO Loops 62
M56800E Target panel 75
M56800E Target Settings panel 41, 44
Output File Name 44
Project Type 43
mac_r 137
Make command 34
makefiles 31
memory address 287
memory directive 287
memory window
P memory 203
X memory 202
menu, command converter server 172
Metrowerks Standard Library (MSL)
for DSP56800E 309
Miscellaneous Trace Buffer Options 344
Miscellaneous Trigger Options 342
modulo addressing
intrinisic functions 157
overview 157
modulo addressing intrinsic functions
__mod_access 160
__mod_error 162
__mod_getint16 161
__mod_init 159
__mod_initint16 160
__mod_setint16 162
__mod_start 160
__mod_stop 161
__mod_update 161
definitions and examples 158
modulo buffer
example 1 163
example 2 164
example diagram 157
modulo buffer API, caveats on using 165
Motorola Documentation 17
msu_r 138
mult 138
mult_r 139

N

negate 124
New Connection window 176
New Project window 26
New window 24
non-volatile registers 83
norm_l 144
norm_s 143
Normal Trigger Modes 335
Normal Unit Action Options 339
number formats, for DSP56800E 80

O

object 289
object keyword 279
off, pragma interrupt 100
on, pragma interrupt 100
optimization
inline assembly 115
pragma, object code 105
optimization_level 104

optimize_iasm 115
optimizing
  code for DSP56800E 95
  register coloring 95
ORG directive 116
  memory space location 116
ORIGIN
  AFTER command 287
  memory address 287
Other Executables settings panel 40
Output Directory field 42

P

P memory 203
  Disassembly Data 205
  Mixed Data 206
  Raw Data 205
  Source Data 206
pad pipeline, debugger 63
Passing Parameters to Functions 82
platform target, defined 16
`pointer` size, ordinal type 80
pointer types (word versus byte pointers) 88
possible hitches
  pragma interrupt 103
Post-Linker list box 42
pragma
  asmoutput 97
  check_c_src_pipeline 98
  check_inline_asm_pipeline 98
  define_section 106
  interrupt 98
  optimization_level 104
  section 108
  unused 109
Pragma Directives
  for DSP56800E 97
pragma interrupt
  alignsp 99
  called 100
  comr 99
  default 100
  off 100
  on 100
  reset 100
  saveall 99
#pragma interrupt and function block 101
#pragma interrupt on | off | reset 101
pragma object code
  optimization 105
pragma optimization 104
pragma optimize_for_size 105
Pre-Linker list box 42
Preprocess command 34
preprocessing 34
  See also IDE User Guide
Program window 196
Project Files versus Makefiles 31
Project Manager window 27
project stationery 26
Project Type 43
projects
  build target 16
  debugging 195
  editing contents of 27
  platform target 16

R

ref_include 271, 275, 289
references, Motorola Documentation 17
register coloring 95
register coloring optimization 95
Register Details Window 201
registers
  display contents 199
  function parameters 82
  general purpose 199
  non-volatile 83
  stack pointer 86
  volatile 83
Related documentation, for targeting manual 17
remote connection
  add a new one 175
  remove an existing one 177
remote connections
  setting up 174
Remote Connections panel 175
Remote Debug Options panel 73
Remote Debugging for the Command Converter Server (local connection) panel 72
Remote Debugging panel 70, 195
Remote Debugging with the Simulator panel 71
remote target board, debugging 177

requirements *See* system requirements
requirements, system 19
reset, pragma interrupt 100
Restoring Target Settings 39
Return Codes 334, 335
Returning Values from Functions 83
ROM copy table 280
ROM to RAM copy 279
rounding 145
runtime initialization 315
Runtime Settings panel 40
runtime, ROM to RAM copy 280

S

sample code
    #pragma interrupt and function block 101
    #pragma interrupt on | off | reset 101
    'called' option in #pragma interrupt 102
    pragma define_section and pragma section 108
    pragma optimization 104
    pragma optimize_for_size 105
    pragma unused 109
    register coloring 95
save/restore registers 182
save/restore registers dialog box 183
saveall, pragma interrupt 99
Saving new target settings
    stationery files 38
section 108
SECTION mapping, in assembly language 116
sections 272, 289, 290
security features, Flash Memory 248
segmentName 290
Set Hardware Breakpoint panel 185
set hardware breakpoints 185
Set Trigger panel 191
set trigger panel 190
set_hfm_base 247
set_hfm_bases 77
set_hfm_config_base 247
set_hfm_erase_mode 77, 248
set_hfm_verify_erase 77, 248
set_hfm_verify_program 77, 248
set_hfmclkd 77, 246
setting breakpoints 197
setting debugger preferences 194
setting up a remote connection 174
setting watchpoints 198
settings panels
    Access Paths 40
    Build Extras 40
    C/C++ Language 45
    C/C++ Warnings 51
    Custom Keywords 40
    Debugger Settings 40
    ELF Disassembler 58
    File Mappings 40
    Global Optimizations 40
    M56800E Assembler 55
    M56800E Linker 65
    M56800E Processor 61
    M56800E Target 43
    M56800E Target panel 75
    M56800E Target Settings panel 41
    Other Executables 40
    Remote Debug Options panel 73
    Remote Debugging panel 70
    Runtime Settings 40
    Source Trees 40
shl 147
shlftNs 148
shlfts 149
`short double` size, floating-point type 81
`short` size, ordinal type 80
Show Addresses and Object Code checkbox 59
Show Code Modules checkbox 59
Show Comments checkbox 60
Show Data Modules checkbox 60
Show Debug Info checkbox 60
Show Headers checkbox 58
Show Relocations checkbox 59
Show Source Code checkbox 59
Show Symbol and String Tables checkbox 58
Show Transitive Closure checkbox 67
shr 150
shr_r 151
shrtNs 152
`signed char` size, ordinal type 80
Simulator 193
sizeof 291
sizeofw 291
small memory model 62

source files, editing 28
Source Trees settings panel 40
special counters 186
S-record 68, 69
S-Record EOL Character list box 69
S-Record, Max Record Length field 69
stack alignment 87
stack frame, for DSP56800E 86
stack pointer register 86
stack size 281
statement-level inline assembly 112
stationery
    saving new target settings 38
stop, intrinsic function 129
storage of code and data for DSP56800E 89
Store Full Path Names checkbox 66
sub 127
support, web page 33
Suppress Warning Messages checkbox 68
symbols, in linker command files 275
syntax, inline assembly language 112
system requirements, Windows 19
system-level connect 245

T

target and debugging panels 169
Target Name field 42
Target Settings panel options
    Linker list box 42
    Output Directory field 42
    Post-Linker list box 42
    Pre-Linker list box 42
    Target Name field 42
Target Settings panels
    Access Paths 40
    Build Extras 40
    C/C++ Language 45
    C/C++ Warnings 51
    Custom Keywords 40
    Debugger Settings 40
    File Mappings 40
    Global Optimizations 40
    M56800E Linker 65
    M56800E Processor 61
    M56800E Target 43
    Other Executables 40
    Runtime Settings 40
    Source Trees 40
Target Settings window 37
This 15
trace buffer 187
Trace Buffer Capture Options 342
Trace Buffer Full Options 343
Trace Buffer Setup panel 188
turn_off_conv_rndg 130
turn_off_sat 130
turn_on_conv_rndg 130
turn_on_sat 131

U

unlock Flash security features 248
`unsigned char` size, ordinal type 80
`unsigned int` size, ordinal type 80
`unsigned long` size, ordinal type 80
`unsigned short` size, ordinal type 80
unused 109
unused code and data, deadstripping 109
Use Extended Mnemonics checkbox 59
using comments in M56800E assembly 115

V

variables, in LCF 275
Verbose Info checkbox 59
viewing memory
    P memory 203
    Register Details Window 201
    X memory 202
volatile registers 83

W

wait 129
watchpoints 198
web site 17
Windows, system requirements 19
writeb 282, 291
writeh 282, 292
writepmem 77
writereg 77
writew 282, 292
writexmem 77

X

X memory 202
XML files
    exporting and importing panel options 38

CWDSP56800E1/D
REV: 1
10/2002